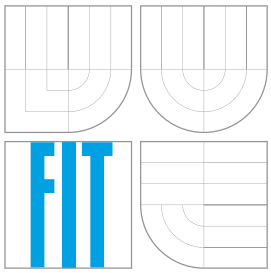
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

OPTIMALIZACE DISTRIBUOVANÉHO KOLEKTORU SÍŤOVÝCH TOKŮ

OPTIMIZATION OF DISTRIBUTED NETWORK FLOW COLLECTOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN WRONA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN ŽÁDNÍK, Ph.D.

BRNO 2016

Zadání diplomové práce/18670/2015/xwrona00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Wrona Jan, Bc.**
Obor: Počítačové sítě a komunikace
Téma: **Optimalizace distribuovaného kolektoru síťových toků**
Optimization of Distributed Network Flow Collector
Kategorie: Algoritmy a datové struktury

Pokyny:

1. Seznamte se s oblastí sběru dat o síťovém provozu na úrovni IP toků. Dále se seznamte s problematikou zpracování velkého množství dat o síťových tocích na distribuovaném kolektoru. Zaměřte se na problematiku spolehlivosti a dostupnosti kolektoru.
2. Analyzujte možnosti rozšíření distribuovaného kolektoru postaveného na nástrojích ipfixcol a fdistdump.
3. Navrhněte rozšíření distribuovaného kolektoru především v oblasti spolehlivosti, dostupnosti a řízení dotazování.
4. Implementujte navržená rozšíření.
5. Proveďte experimenty s kolektorem za účelem evaluace implementovaných rozšíření.
6. Zhodnoťte dosažené výsledky a diskutujte možná další rozšíření.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Žádník Martin, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

Kotásek

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Tato práce se zabývá optimalizací distribuovaného kolektoru informací o IP tocích. V současnosti často využívaným řešením je centralizovaný kolektor, který ale naráží na své výkonnostní limity v prostředí rozsáhlých a vysokorychlostních sítí. Implementace distribuovaného kolektoru je teprve v počátcích a je potřeba hledat řešení, která dokážou plně využít potenciál distribuovaného systému. Proto práce přináší návrh architektury bez sdílených komponent a bez jediného bodu selhání, distribuovaný kolektor je s jejím použitím odolný proti výpadku minimálně jednoho uzlu. Součástí práce je také distribuovaný dotazovací systém, jehož výkon škáluje lineárně v závislosti na počtu uzlů.

Abstract

This thesis is focused on the optimization of distributed IP flow information collector. Nowadays, the centralized collector is a frequently used solution but is already reaching its performance limits in large scale and high-speed networks. The implementation of the distributed collector is in its early phase and it is necessary to look for solutions that will use it to its full potential. Therefore this thesis proposes a shared nothing architecture without a single point of failure. Using the above proposed architecture, the distributed collector is tolerant to the failure of at least one node. A distributed flow data analysis software, whose performance scales linearly with the number of nodes, is also part of this thesis.

Klíčová slova

NetFlow, IPFIX, kolektor toků, síťové toky, distribuovaný systém, vysoká dostupnost, odolnost proti poruchám, Corosync, Pacemaker, GlusterFS, IPFIXcol, fdistdump

Keywords

NetFlow, IPFIX, flow collector, network flow, distributed system, high availability, fault tolerance, failover, Corosync, Pacemaker, GlusterFS, IPFIXcol, fdistdump

Citace

WRONA, Jan. *Optimalizace distribuovaného kolektoru síťových toků*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Žádník Martin.

Optimalizace distribuovaného kolektoru síťových toků

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Martina Žádníka, Ph.D.

.....
Jan Wrona
25. května 2016

Poděkování

Děkuji panu Ing. Martinu Žádníkovi, Ph.D. za odborné vedení mé práce, poskytnutí množství cenných rad, připomínek a návrhů.

Obsah

Kapitola 1

Úvod

Sledování počítačových sítí poskytuje administrátorům data, která jsou potřeba k jejich správě a optimalizaci. Typicky se k tomu používají protokoly SNMP, RMON, častým řešením je také monitoring IP toků [54]. Poslední zmiňované se sestává z několika fází: záchyt paketů, měření a export toků, sběr dat a jejich analýza [35]. Fyzicky se celý řetězec typicky vměstná do dvou komponent, kterými jsou měřící sonda (nazývána také exportér) a kolektor. První dvě fáze jsou prováděny na měřící sondě, ze které jsou záznamy o tocích zasílány na kolektor. Úkolem kolektoru je data uložit a umožnit nad nimi provádět analýzu, ať už automatickou nebo manuální prováděnou správcem. Kolektor hraje významnou roli také při dohledávání incidentů a kybernetických útoků. Z výše uvedeného je patrné, že kolektor je centrální bod celého monitorovacího řetězce a může se snadno stát úzkým hrdlem. Již nyní se tak děje na vysokorychlostních sítích, kde současně produkuje záznamy několik měřících sond.

Jako hlavní problém současných řešení kolektoru se zdá být nízká propustnost vstupně výstupního subsystému spolu se špatnými možnostmi v oblasti paralelizace [41, 45]. Přitom rychlost čtení záznamů o tocích z permanentního úložiště je pro kolektor naprosto klíčová. Objem jak ukládaných tak analyzovaných dat roste, centralizovaný kolektor ale nenabízí odpovídající úroveň škálovatelnosti. Druhý problém je dlouhá doba odezvy na události v síti, protože na exportéru musí dojít k expiraci toku, kolektor musí data přijmout, uložit, a až potom je možné provést jejich analýzu [69].

Abychom vyhověli dnešním požadavkům, je potřeba monitorovací systém, který může růst spolu s nároky, které na něj jsou kladeny. Odpovídající míru horizontální škálovatelnosti výpočetního výkonu i úložného prostoru poskytuje dnes tak oblíbený pojem cloud computing [7]. Za tímto populárním souslovím ale nestojí nic jiného, než vysoce dostupný a spolehlivý distribuovaný systém. Potřeba lepší škálovatelnosti vedla ke vzniku konceptu nasazení kolektoru na distribuovaný systém [69]. Jeho návrh ale momentálně neřeší důležitý aspekt vysoké dostupnosti a spolehlivosti, zároveň je potřeba jej optimalizovat po stránce ukládacího a dotazovacího výkonu. Při poruše některé jeho komponenty nesmí dojít k výpadku systému jako celku a nesmí dojít ke ztrátě dat. Začínají se tak propojovat doposud nesouvisející oblasti: kolektor záznamů o síťových tocích, distribuované systémy a vysoká dostupnost. Jejich studium je prvním bodem této práce, od kterého se odvíjí další postup v podobě hledání průniku a možnosti kombinace zmíněných oblastí. Tato snaha ústí v návrh rozšíření distribuovaného kolektoru, který poskytuje vysokou dostupnost, spolehlivost a optimalizované řízení dotazování. Dalším bodem je implementace návrhu, po které následuje řada experimentů prováděných za účelem vyhodnocení návrhu a nalezení případných možností na pokračování v této práci.

Text práce je rozdělen do několika logických celků. Kapitola 2 obsahuje informace o správě počítačových sítí a sledování síťového provozu, včetně důvodů a cílů těchto činností. Rozebrány jsou možnosti od nejzákladnějších, v podobě SNMP, až po pokročilou hloubkovou analýzu paketů. Největší důraz je kladen na monitorovací metodu sběru dat na úrovni IP toků, která je pro tuto práci klíčová a proto se jí věnuje celá kapitola 3. Detailně se rozebírají fáze pro záchyt a předzpracování paketů na sondě, měření a export vzniklých záznamů, sběr a uložení dat na kolektoru a také poslední fáze v podobě analýzy dat. Kapitola 4 nabízí teoretický, ale i praktický pohled na obecný distribuovaný systém a zabývá se také problematikou vysoké dostupnosti. Na základě předchozí analýzy byl vytvořen návrh požadovaných rozšíření spojující různé techniky a přístupy, pomocí kterých můžeme dosáhnout zadaných cílů. V kapitole 5 jsou proto představeny dvě architektury obohacující současný stav, první se vyznačuje vysokým výkonem a vysokou cenou, druhá cílí na nízký počet potřebných komponent a tedy i nižší cenu. Navazující kapitola 6 popisuje implementaci jedné z představených architektur pomocí programového zásobníku o několika vrstvách. Potřebné nastavení všech programů včetně útržků důležitých částí konfigurace popisuje kapitola 7. Po zkušebním zapojení nové architektury do provozu bylo možné provést sadu několika experimentů, které jsou spolu s jejich výsledky uvedeny v kapitole 8. Následuje už jen kapitola 9, která obsahuje závěrečné zhodnocení práce a diskuzi o možnostech dalšího vývoje.

Kapitola 2

Správa sítě a sledování síťového provozu

Většina modelů životního cyklu počítačové sítě v některé své fázi zahrnuje správu sítě a monitorování provozu. Příkladem může být model PPDIOO [30], který se skládá z přípravy, plánování, návrhu, implementace, provozování a optimalizace. Právě fáze provozování zahrnuje udržování sítě v provozuschopném stavu, zajištění vysoké dostupnosti, detekci a korekci chyb, správu bezpečnosti, monitorování výkonnosti a provozu. V celém životním cyklu je tato fáze časově nejdelší, přesto ale nepřináší žádné výrazné zásahy do navržené architektury či změny topologie. Její význam ale přesto není zanedbatelný. Informace získané dlouhodobým sledováním provozu, výkonu, bezpečnosti, dostupnosti služeb a mnoha dalších faktorů můžeme s výhodou využít ve fázi následující, tedy při optimalizaci. Cílem optimalizace je identifikace a korekce problémů před tím, než ovlivní bezproblémový běh sítě. Toho lze ale dosáhnou pouze proaktivní správou s využitím informací získaných během provozní fáze. Správa a monitoring jsou tedy nezbytnou, přesto často zanedbávanou součástí návrhu a následného provozu počítačových sítí.

Sledováním sítě typicky rozumíme sledování stavu jednotlivých komponent [53]: směrovače, přepínače, servery a podobně. Zajímá nás, zda je vůbec zařízení v operativním stavu, zda je schopno zpracovávat požadavky, jaké jsou stavy čítačů na jednotlivých rozhraních a mnoho dalších stavových informací. V případě, že je zařízení nastaveno tak, že tyto informace sděluje autonomně, bez jakéhokoliv zásahu administrátora, jedná se o monitorování pasivní. Tento způsob zahrnuje synchronní a asynchronní hlášení. První zmiňované obsahuje např. pravidelné hlášení o stavu zařízení z důvodu tvorby statistik a grafů, druhé zmiňované je typicky zapříčiněno nepravidelnou událostí či vstupem do chybového stavu.

Sledování síťového provozu je od sledování sítě rozdílné v tom, že více než stav zařízení nás zajímají data, která přes zařízení v síti tečou. Zde se otevírají poměrně široké možnosti: máme na výběr kde, jakým způsobem a do jaké hloubky budeme provoz sledovat. Můžeme data uchovávat, nebo pouze provádět analýzu proudově v reálném čase. V případě, že chceme data uchovávat, můžeme volit míru agregace od vysoké (pouze statistické informace), přes střední (informace o tocích) až po žádnou (uchovávají se celé rámce). Některé možnosti s sebou ale přináší také řadu problémů. V případě analýzy v reálném čase můžeme narazit na nedostatek výpočetního výkonu. Aplikovat některé algoritmy, např. pro přesnou identifikaci aplikačních protokolů v reálném čase na vysokorychlostních sítích zkrátka není v dnešní době možné [65]. Uchovávání celých rámců bez agregace naopak klade vysoké nároky na úložiště. S příchodem hloubkové analýzy paketů nově také narážíme na problémy

netechnického charakteru související s narušením soukromí uživatelů internetu. Mimo využití v bezpečnosti a optimalizaci sítě totiž tato metoda přináší možnosti jako pokročilé dolování dat, cenzuru, nebo porušování principu síťové neutrality [5].

Doposud byla řeč o pouhém pozorování, tedy o pasivním monitoringu provozu. I zde se ale nabízí možnost monitoringu aktivního, při kterém monitorujeme síť stejným způsobem, navíc ale do sítě určitá data vkládáme. Touto metodou lze snadno měřit výkonnost jednotlivých komponent a sítě jako celku. Můžeme měřit obousměrné zpoždění (round-trip time, RTT), odezvu aplikačního serveru na požadavek, propustnost nebo jitter [27]. Mezi využití aktivního sledování patří také verifikace nastavení kvality služeb QoS.

2.1 SNMP, RMON

V oblasti správy a sledování sítě existuje celá řada nástrojů a protokolů. Často využívaným řešením je IETF standardizovaný protokol Simple Network Management Protocol, SNMP [14]. Jeho jádro tvoří síťová zařízení, množina jednoduchých operací a hierarchická databáze. Síť spravovaná pomocí SNMP se typicky skládá ze tří klíčových **komponent** [53]:

Spravované zařízení: směrovač, přepínač, tiskárna, server s libovolným OS.

Agent: software běžící na spravovaném zařízení.

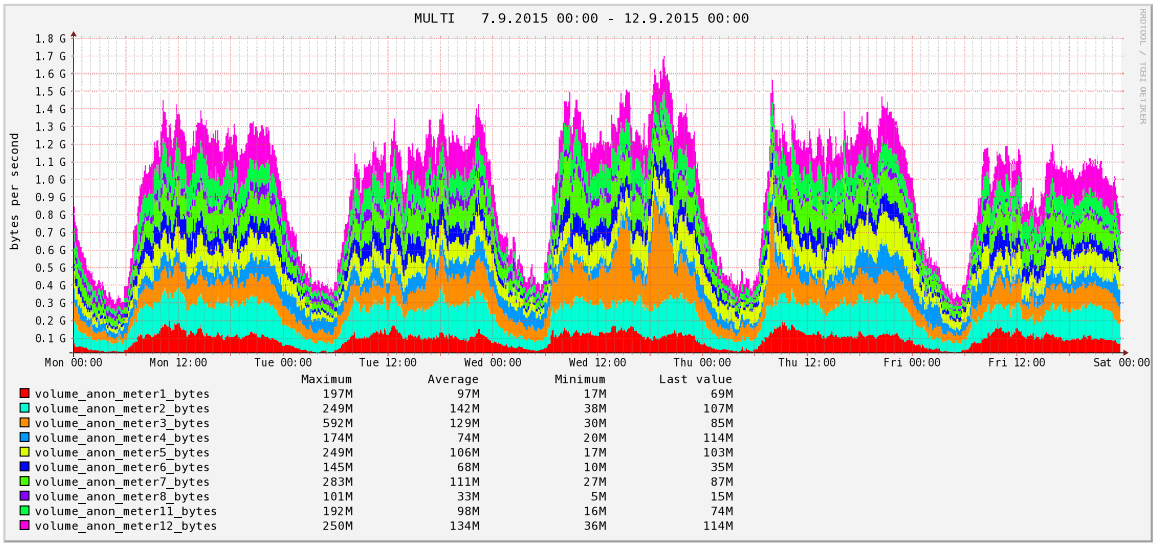
Network management station (NMS): software běžící na správcovské stanici.

Správcovská stanice má za úkol monitorovat a spravovat množinu zařízení na počítačové síti. Jak tato stanice, tak i spravovaná zařízení musí být vybaveny příslušným software, který vzájemnou interakci zprostředkuje. Zařízení, jenž mohou zastávat roli agenta, existuje široké spektrum. Implementuje jej většina operačních systémů síťových prvků, existuje množství implementací pro běžně používané operační systémy, SNMP agenta můžeme ale nalézt třeba v tiskárnách nebo databázích[1] (nemusí se vždy jednat o fyzická zařízení). Takto vybavená zařízení potom mohou zpracovávat příkazy přijaté od NMS. Správcovská stanice může s agenty komunikovat dvěma způsoby. První, synchronní, zahrnuje PDU typu `Get` a `Set`. Metodami typu `Get` je možné vyčíst od agenta hodnotu nějaké proměnné nebo seznam proměnných. Metodou `Set` se naopak provádí změna hodnoty proměnné nebo seznamu proměnných na straně agenta. Agent na obě metody odpovídá metodou `Response` obsahující původní hodnotu při `Get` a hodnotu novou při metodách `Set`. Druhým způsobem komunikace jsou PDU `Trap`. Tyto asynchronní notifikace vznikají na straně agenta a umožňují informovat správce o nějaké nečekané události. Zpráva si s sebou nese všechny potřebné informace jako systémový čas nebo identifikátor události, aby bylo na správcovské stanici možné rozpoznat k jaké změně nebo chybě na zařízení došlo.

SNMP samotné ale nedefinuje, které proměnné je možné vyčítat nebo nastavovat. Využívá k tomu hierarchickou stromovou databázi zvanou **Management Information Base**, MIB. Každý záznam v databázi má unikátní *identifikátor objektu* a právě ten se vyskytuje ve všech typech PDU. Např. při metodě `Get` agent vyhledá tento identifikátor v MIB a pokud jsou splněny určité požadavky (zařízení tuto proměnnou nabízí a žadatel má oprávnění ji číst), odpoví metodou `Response`.

Jak ale souvisí protokol SNMP s monitorováním sítě? Lze pomocí něj vůbec monitoring provádět? Odpověď zní ano, ale pouze do určité míry. Základní přehled o provozu tekoucím přes zařízení lze zjistit například z hodnot čítačů na síťových rozhraních [53].

[1] https://github.com/masterzen/mysql-snmp



Obrázek 2.1: Graf vykreslený z dat round-robin databáze zobrazující množství přenesených bajtů za sekundu na několika linkách během za období pěti dnů.

Periodickým vyčítáním takových hodnot tak lze tvořit například přehledové grafy jako je vidět na obrázku 2.1. To ale v mnoha případech není dostačující, míra informace v takových datech je poměrně nízká. Potřeba monitorovat síť jako celek, nikoliv pouze jednotlivá zařízení, je jeden z důvodů vzniku protokolu Remote Network Monitoring, **RMON** [63]. Jeho původní verze pracovala s první druhou vrstvou ISO/OSI modelu, v pozdější verzi byla podpora rozšířena na třetí vrstvu, je ale možné pracovat až s vrstvou aplikační. Jednotlivé komponenty specifikace RMON jsou podobné SNMP, místy se liší terminologie (spravované zařízení s agentním software se nazývá sonda, není to ale jediný rozdíl). Do sondy se částečně přesunula inteligence, která musela být u SMTP implementována ve správcovské aplikaci, takže sonda u RMON sbírá informace, analyzuje pakety a je schopná monitorovat celý síťový segment. Funguje do jisté míry autonomně, čímž je redukován provoz, který u SNMP produkovalo neustálé vyčítání hodnot. Sonda v architektuře figuruje jako server, naopak správcovská aplikace jako klient, který si vyčítá data naměřené a zpracované sondou. Historická i aktuální data o provozu segmentu se tak mohou vyčíst až ve chvíli, kdy je potřeba je zobrazit. Z těchto vlastností plyne ale jedna nevýhoda: RMON je pro síťovém zařízení podstatně větším břemenem. Například u méně výkonného směrovače tak může nastat situace, kdy kvůli monitoringu nebude mít dostatek zdrojů na to, aby mohl plnit svůj primární účel. Principiálně je ale RMON blíže technikám založeným na monitorování síťových toků, které jsou rozebrány v kapitole 3.

2.2 Hloubková analýza paketů

Hloubková analýza paketů, označovaná zkratkou DPI podle anglického Deep Packet Inspection, je technika, kterou je možné využít jak k monitorování a analýze, tak k filtraci síťového provozu [8]. Od doposud zmiňovaných metod se zásadně liší ve dvou přístupech. Monitorování toků využívá ke svému fungování různé statistické údaje, časové značky, hodnoty čítačů a také data z hlaviček protokolů. Jeho primárním zdrojem informací tak jsou především metadata paketů v podobě internetových adres, čísel portů nebo třeba příznaků

protokolu TCP. Jak již název napovídá, DPI se v tomto ohledu liší v tom, že jeho primárním zdrojem informací nejsou metadata, ale samotná přenášená data. Další zásadní rozdíl tkví v agregaci. Export toků je založen na slučování paketů (právě do toků), kdežto DPI typicky žádnou agregaci neprovádí a soustředí se na každý jednotlivý paket. Z těchto důvodů jsou ostatní způsoby monitorování považovány za bezpečnější a méně náchylné na problémy s porušováním soukromí.

V různých formách je DPI již běžně nasazované a ukazuje se jako velice slibná technologie v oblasti obrany proti kybernetickým hrozbám. Je možné ji **využít** k hledání virů, spamu nebo třeba nevalidních hodnot na úrovni aplikačních protokolů [65]. Poslední zmiňovaný způsob použití uvádím v souvislosti se zranitelnostmi jako Heartbleed nebo Shellshock, které je možné detekovat právě technikou hloubkové analýzy paketů. Efektivní je ale také při identifikaci buffer overflow, denial of service (DoS), SQL injection, cross-site scripting nebo jiných útoků na aplikační úrovni [47]. Proto je hloubková analýza součástí mnoha IDS a IPS systémů, jako např. Snort[2] nebo Bro[3].

Tato technologie je také mocný nástroj v rukou ISP a vlád, které ji mohou využít pro odposlechy, cílenou reklamu, uplatnění autorských práv, nebo dokonce k cenzuře [9]. Je známo, že za účelem **cenzury** internetového obsahu využívá DPI vláda Čínské lidové republiky [67], pokud je obsahem paketu zakázané klíčové slovo, spojení je přerušeno. Tyto praktiky, spadající pod projekt označovaný jako Zlatý Štít, tak ze sítí čínských ISP znesnadňují přístup například k webovému obsahu Googlu, Wikipedie nebo Facebooku.

Ani hloubková analýza paketů ovšem není všemocná. Jeden ze způsobů, kterými je možné ji efektivně **znemožnit**, je šifrování. Ačkoliv existují metody, které dokáží rozpoznat aplikační protokol pomocí statistických signatur provozu i přes jeho šifrování [65], nejedná se zdaleka o tak citlivou informaci, jakou je samotný datový obsah. Metody fragmentace a segmentace sice dokáží hloubkovou analýzu ztížit (je potřeba fragmenty spojit nebo si pamatovat stavové informace), nikoliv ale zcela znemožnit. Nevýhodou jsou také poměrně vysoké nároky, které DPI klade na výpočetní prostředky. Záleží samozřejmě na druhu analýzy, ale hloubkově prozkoumat každý paket na vysokorychlostních sítích může být problém.

[2] https://www.snort.org/
[3] https://www.bro.org/

Kapitola 3

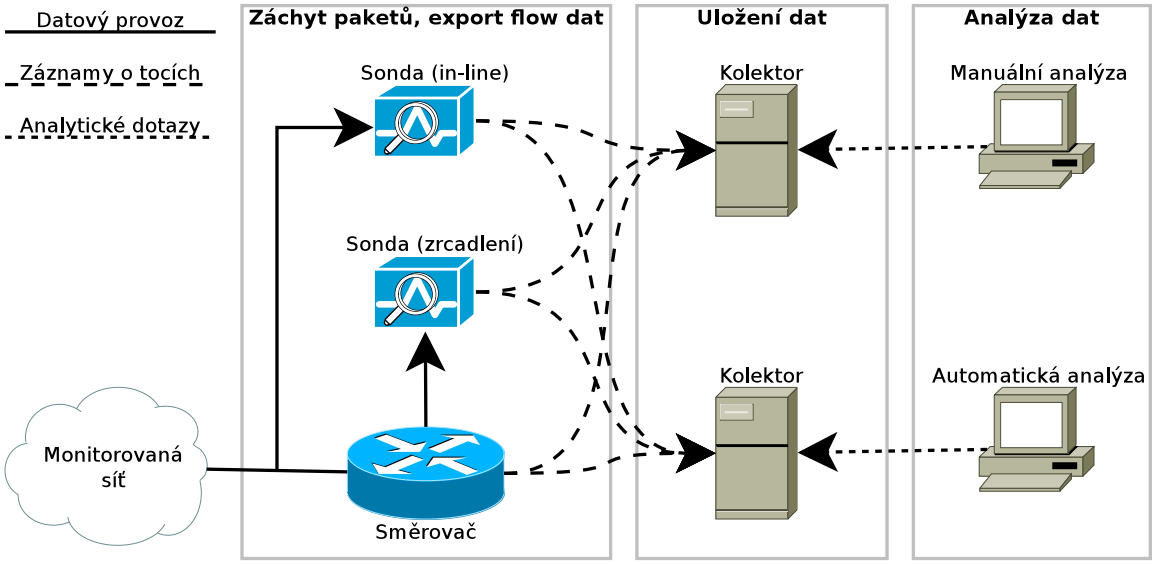
Monitorování síťových toků

Přístupy z předchozí kapitoly byly uvedeny z důvodu zařazení práce do širšího kontextu, tato kapitola se zabývá metodou monitorování síťových toků, která je pro práci klíčová. V kapitole se objevuje několik **pojmů**, které je potřeba definovat a objasnit. První takový pojem je už v názvu metody, tedy *síťový tok*. Jak uvádí [19], definic existuje několik, v této práci se ale bude vždy jednat o množinu paketů nebo rámců procházející pozorovacím bodem v síti v určitém časovém intervalu. Všechny pakety patřící do určitého toku musí mít shodnou množinu předem určených vlastností. Které vlastnosti do této množiny typicky v praxi patří je uvedeno v sekci 3.2. Dalším často používaným pojmem v této oblasti je *NetFlow*. Ačkoliv se v praxi i v literatuře můžeme setkat s různými asociacemi s tímto pojmem, v této práci NetFlow označuje schopnost sběru a exportu záznamů o tocích, která byla poprvé implementována do směrovačů společnosti Cisco. *IP Flow Information Export (IPFIX)* je také často skloňovaným pojmem, jedná se o IETF standardizovaný protokol [19] určený pro výměnu informací o tocích. Pokud není uvedeno jinak, zdrojem informací v této kapitole je [35].

O **oblíbenosti** tohoto způsobu monitorování svědčí například průzkum [54] mezi ISP z roku 2013, kde více než 70 % respondentů uvedlo, že disponují technikou umožňující export flow dat. Výsledky průzkumu nasvědčují tomu, že mnozí výrobci síťového hardwaru již do svých zařízení tuto technologii implementovali. U takových výrobků může být funkcionalita označena jako NetFlow ale i jinak, např. J-flow u Juniper Networks[1], NetStream u HP, Traffic Flow u MikroTik. Rozsáhlá integrace této funkcionality do operačních systémů směrovačů, přepínačů nebo firewallů tak dovoluje provozovatelům sítí provádět významnou část celého monitorovacího procesu s použitím již existujícího hardware, tedy bez dalších finančních nákladů. Existují ale mnohé další důvody nasazení právě tohoto přístupu. Jedním z nich je povinnost evropských ISP zaznamenávat data o spojení po dobu šesti měsíců až dvou let z důvodu možného pozdějšího vyšetřování zločinu [26]. A právě požadavky, které tato povinnost specifikuje umožňují použít sledování a zaznamenávání toků pro její splnění. Další výhoda oproti např. DPI je fakt, že záznamy jsou konstruovány primárně z hlaviček protokolů a záznamy o tocích tak obsahují méně citlivých informací. S rostoucí mírou vkládání aplikačních dat také do flow záznamů se ale tato výhoda postupně vytrácí.

Typická architektura měření síťových toků se skládá z **několika komponent a fází**. Záchyt a předzpracování paketů na síťovém zařízení (směrovač, exportér, ...) je první fází, popsána je v sekci 3.1. Proces samotného měření, agregace do toků a následného exportu vzniklých dat je popsán v sekci 3.2. Třetí fáze, uvedená v sekci 3.3, se zabývá příjmem expor-

[1] http://www.juniper.net/us/en/local/pdf/app-notes/3500204-en.pdf



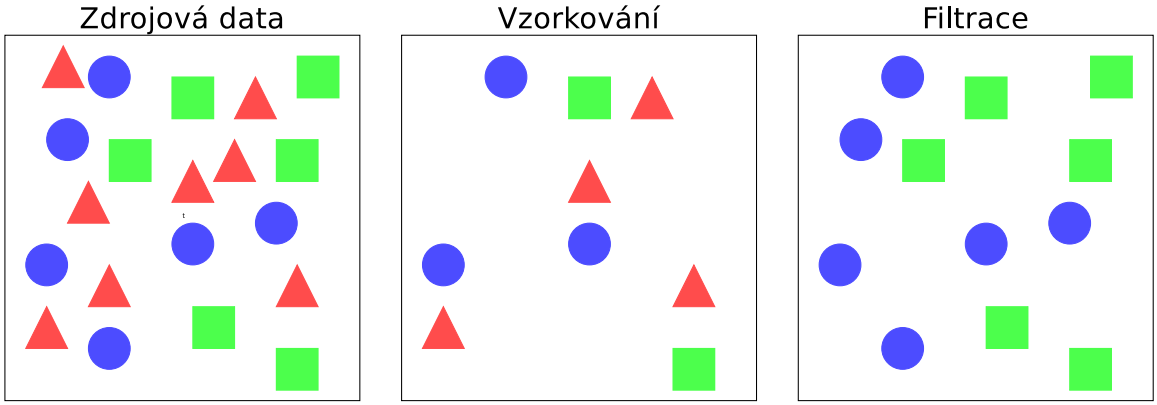
Obrázek 3.1: Znázornění typické architektury měření IP síťových toků.

tovaných dat, zpracováním a hlavně uložením do volatilního nebo perzistentního úložiště. Poslední fází architektury je offline analýza uložených dat, manuální nebo automatizovaná, věnuje se jí sekce 3.4. Celkový pohled na architekturu je zobrazen diagramem 3.1. Ačkoliv by bylo možné všechny fáze provádět na jednom systému, v praxi je architektura často nasazená na systém distribuovaný. Jedním z důvodů je zajištění dostatečného výpočetního a/nebo vstupně výstupního výkonu, druhým je potom fyzické umístění pozorovaných bodů. V rámci jednoho monitorovacího systému je potom možné měřit více různých bodů v síti, které díky svému zeměpisnému rozmístění nemohou být měřeny jedním fyzickým systémem současně.

3.1 Záchyt a předzpracování paketů

Mimo záchytu paketů z linky se tato fáze zabývá také jejich předzpracováním, aby se co nejdříve zmenšil objem dat proudících z jedné fáze do druhé. Tato sekce popisuje také zařízení, kterými je možné záchyt provádět a různé způsoby jejich zapojení do sítě.

Je zřejmé, že pakety či rámce je nejprve potřeba nějakým způsobem z monitorované linky **získat** (zachytit), ideálně tak, aby provoz na lince nebyl ovlivněn. Typicky je tento bod obstaráván síťovou kartou (Network Interface Controller, NIC), která se může ale nemusí lišit od běžné karty, kterou jsme zvyklí používat v PC. Vzápětí se každý přijatý paket obohatí o časovou značku. Tato akce je důležitá především z pohledu pozdější analytické fáze, kde přesnost časových značek hraje významnou roli. Různé způsoby vkládání (hardware vs. software) ale mají různé přesnosti. Zde se projevuje první výhoda použití specializovaných NIC určených pro monitoring, které nabízejí přesnost výrazně vyšší (až jednotky nanosekund) [49], než softwarové řešení. Pro agregaci paketů do záznamů o tocích jsou potřeba pouze informace z hlaviček protokolů umístěné na začátku paketu. Pokud nechceme dále zpracovávat aplikační data, je možné paket oříznout a zachovat pouze prvních několik bajtů. Pokud budeme uvažovat nejčastější situaci, potřebujeme zachovat hlavičku Ethernetového rámce (14 bajtů), IPv4 protokolu (20 bajtů) a TCP protokolu (20 bajtů). V případě MTU (Maximum Transmission Unit) o velikosti 1500 bajtů je tak možné za-



Obrázek 3.2: Rozdíl mezi vzorkováním a filtrováním. Vzorkování přibližně v poměru 1:3, filtrace jednoho prvku (červený trojúhelník).

nedbat až 96 % paketu, čímž se výrazně redukuje tok dat v systému a sníží se množství potřebné paměti.

Posledním bodem v této fázi je **vzorkování a filtrování paketů** (zobrazeno na obrázku 3.2). Cílem vzorkování je zpracovat pouze podmnožinu všech paketů, ale zároveň mít stále možnost odhadnout nebo dopočítat vlastnosti celé původní množiny [38]. Důvodem zavedení vzorkování jsou nižší nároky kladené na následující fáze. Čím nižší vzorkovací frekvence, tím méně paketů se bude zpracovávat, znamená to ale také menší dosaženou přesnost. Vzorkování obecně je možné provádět dvěma způsoby: systematicky a náhodně. Systematické vzorkování volí prvky na základě deterministické funkce, v praxi se tak může jednat třeba o výběr každého třetího paketu (vzorkovací poměr 1:3). Výskyt paketů a jejich vlastností na síti je ale stochastický proces, použití systematického vzorkování tak vždy znamená nebezpečí zavedení nežádoucího zkreslení. Náhodné vzorkování naopak volí prvky na základě náhodné funkce. Existuje řada způsobů náhodného vzorkování: uniformní, neuniformní, n z N apod., obecně ale platí, že tyto metody více vyhovují charakteristice paketů na síti. Náhodné vzorkování tak oproti systematickému může přinést vůbec anebo výrazně méně korelované výsledky [68]. Cílem filtrování je deterministický výběr paketů v závislosti na jeho obsahu. Na takto vybrané pakety je následně aplikovaná akce propuštění nebo zahození. Rozdíl oproti vzorkování je tak v tom, že nezávisí na pozici příjmu paketu v čase, ale zda jeho obsah splňuje nebo nesplňuje jednu či více podmínek. I filtrování je možné ještě dále dělit. Záleží zda chceme volit podmnožinu na základě hodnoty určité vlastnosti paketu, např. IP adresy, nebo na základě výsledku hašovací funkce aplikované na celý paket nebo jeho část. První zmíněnou lze filtrovat např. podle rozsahu určité hodnoty, druhá metoda je vhodná především pro výběr paketů s jednou či více shodnými hodnotami.

Většina **zařízení pro záchyt paketů** je určena pro drátové sítě, které svým rozsahem mohou sahat od malých LAN až po páteřní spoje s mnohonásobně větší propustností. Stejným způsobem je ale možné monitorovat jak bezdrátové, tak virtuální sítě [17]. V případě drátových sítí, kde je monitorování nejčastější, se nabízí několik možností realizace této fáze (všechny jsou zobrazeny na obrázku 3.1). Pokud je záchyt a export prováděn na zařízení, které má implicitně přístup k paketům (směrovač, přepínač, nebo třeba firewall), je situace jednoduchá. Běžně se ale tyto akce provádí na specializovaném zařízení zvaném sonda nebo exportér. V takovém případě je ale potřeba data do sondy nějak dostat. První možností je použití externího hardwaru, který zajistí duplikaci veškerého procházejícího provozu. Je možné jej provozovat jak na optických tak na metalických sítích, navíc typicky nezpůsobuje

žádnou odezvu ani modifikaci dat. Druhou možností je zrcadlení portů. Tato varianta je levnější, není totiž třeba dodatečný hardware. Pro zařízení, která musí zrcadlení provádět to ale znamená práci navíc. Může se tak stát, že např. na směrovači se díky zrcadlení portů zvýší využití procesoru natolik, že to ovlivní samotný sledovaný provoz. Bylo také experimentálně zjištěno, že zrcadlení portů může zvýšit odezvu, jitter nebo dokonce změnit pořadí paketů [66].

Programy a knihovny umožňující záchyt, ořezání a filtraci paketů existují pro všechny rozšířené operační systémy. Často používanou knihovnou je libpcap, která je součástí analyzátoru paketů zvaného tcpdump[2]. Při použití běžné síťové karty je ale propustnost paketů do uživatelského paměťového prostoru silně závislá na implementaci síťového subsystému jádra operačního systému, který ale typicky není stavěný na podobné použití a při rychlostech dnešních linek běžně dosahujících 100 Gbps je propustnost nedostačující. Zde se opět projevuje výhoda specializovaných NIC, které dokáží přenášet pakety do uživatelské paměti pomocí přímého přístupu do paměti, tedy bez průchodu pomalým operačním systémem [49].

3.2 Měření a export toků

Vstupem pro tuto fázi jsou již dříve předzpracované pakety. Ty jsou v závislosti na zvoleném *agregačním klíči* slučovány do záznamů o tocích. Dočasné úložiště pro tyto záznamy se nazývá *mezipaměť toků* (*flow cache*) a typicky umožňuje záznam přidat, aktualizovat a odebrat. Ještě před odesláním záznamů směrem ke kolektoru je možné provést další vzorkování a filtraci [38]. Z důvodu zvýšení efektivity a interoperability se před přenosem provede zapouzdření do aplikačního protokolu (řeč je o NetFlow v5 [16], NetFlow v9 [15] IPFIX aj.). Kvůli proudovému charakteru vzniklého provozu ale není ani volba transportního protokolu zcela jednoznačná. Nabízí se TCP, UDP a SCTP, výhody a nevýhody ve spojení s přenosem záznamů ke kolektoru jsou popsány níže.

Jak bylo uvedeno dříve, všechny pakety patřící do určitého toku musí mít shodnou množinu vlastností. Těmito vlastnostmi se typicky rozumí hodnoty určitých polí síťového a transportního protokolu, meze se ale v tomto směru nekladou. Ostatní vlastnosti toku je možné zaznamenávat také, ty už ale nemusí být u všech paketů daného toku shodné. První množina vlastností se nazývá klíčová, druhá je neklíčová a jde o základní stavební kámen záznamu o toku. Zde narážíme na velkou nevýhodu staršího protokolu NetFlow v5, který jednoznačně definuje podobu záznamu, tedy jednotlivé elementy i jejich pořadí[3]. Proto NetFlow v5 nepodporuje ani IPv6 adresy, které jsou delší a do pevně dané struktury se nevejdou. Lépe na tom jsou NetFlow v9 a IPFIX, které podporují variabilní podobu záznamu o toku. Elementy podporované protokolem IPFIX se nazývají **informační elementy** [19] a jejich databáze[4] je spravována autoritou IANA, je ale možné používat také informační elementy, které nejsou zaznamenány ve zmíněné databázi. Jedná se o enterprise položky, u kterých ale není zaručena kompatibilita mezi výrobci. Z toho důvodu by se měly primárně používat elementy obsažené v IANA databázi, u kterých by kompatibilita zaručena být měla.

Tabulka 3.1 zobrazuje některé (často používané) IPFIX informační elementy. Ačkoliv je v tabulce nejvyšší vrstva ISO/OSI modelu transportní, díky enterprise položkám je

[2] http://www.tcpdump.org/

[3] http://www.cisco.com/c/en/us/td/docs/net_mgmt/netflow_collection_engine/3-6/user/guide/format.html

[4] http://www.iana.org/assignments/ipfix/ipfix.xhtml

Vrstva	Název	Popis
Linková	sourceMacAddress	IEEE 802 zdrojová MAC adresa.
	dot1qVlanId	Hodnota VLAN identifikátoru z Ethernetového rámce.
Síťová	sourceIPv4Address	IPv4 zdrojová adresa z hlavičky IP paketu.
	sourceIPv6Address	IPv6 zdrojová adresa z hlavičky IP paketu.
	protocolIdentifier	Hodnota protokolového čísla z hlavičky IP paketu.
Transportní	sourceTransportPort	Identifikátor zdrojového portu z hlavičky transportního protokolu.
	tcpControlBits	Kontrolní bity protokolu TCP.
Žádná	octetDeltaCount	Počet oktetů příchozích paketů toku, včetně IP hlavičky.
	flowStartMilliseconds	Časová značka prvního paketu v toku.

Tabulka 3.1: Ukázka některých IPFIX informačních elementů.

možné zahrnout až aplikační vrstvu a kombinovat tak měření na úrovni toků s hloubkovou analýzou paketu, viz 2.2. Kromě elementů, které extrahují hodnotu z hlavičky nebo obsahu rámce/paketu existují také elementy s jinou sémantikou [18]. V tabulce uvedený `octetDeltaCount` představuje čítač, který po zařazení paketu do toku inkrementuje svou hodnotu o počet oktetů paketu. Při exportu tak obsahuje velikost celého toku v oktetech, což je hodnotný statistický údaj. Druhý takový, `flowStartMilliseconds`, obsahuje časovou značku příchodu prvního paketu v toku, tedy čas vzniku záznamu. Možnosti, které v tomto směru nabízí protokol IPFIX, jsou skutečně bohaté.

Po příchodu paketu na exportér jsou zjištěny hodnoty všech požadovaných informačních elementů. Některé jsou vyčteny z paketu (linkové, síťové, transportní adresy, ...), jiné jsou doplněny exportérem (časové značky, hodnoty čítačů, ...). Dále přichází na řadu operace nad datovou strukturou **mezipaměti toků**. Jde o tabulku, ve které jsou na exportéru uchovány záznamy o tocích od doby vzniku záznamu až po jeho expiraci. Množina klíčových elementů pro tuto tabulku představuje obdobu primárního klíče v relačních databázích, kombinace hodnot klíčových elementů tak musí být unikátní v rámci celé mezipaměti. Když jsou vyčteny všechny hodnoty klíčových elementů z příchozího paketu, je provedeno jejich vyhledání v mezipaměti. Pokud není nalezena shoda, jedná se o první paket toku a v mezipaměti je vytvořen nový záznam. Množina klíčových hodnot tvoří identifikátor, množina neklíčových hodnot má za cíl vystihnout charakteristiku toku. Pokud ale shoda nalezena je, existující záznam se pouze aktualizuje tak, že klíčové položky záznamu jsou ponechány beze změny a neklíčové jsou upraveny dle sémantiky elementu. Např. čítače jsou inkrementovány, časové značky jsou aktualizovány a podobně.

Jak bylo uvedeno na začátku této kapitoly, všechny pakety patřící do určitého toku musí mít shodnou množinu předem určených vlastností. Tato množina není nic jiného než klíčové elementy, nalezneme pro ni také název **agregační klíč**. Zkoumáním různých exportérů zjistíme, že množina elementů tvořících agregační klíč není unifikovaná. Typicky však obsahuje zdrojovou a cílovou IP adresu, zdrojový a cílový port a IP protokol. Některé softwarové exportéry (ipt-NetFlow[5], nProbe[6]) ještě přidávají VLAN ID nebo IP Type of

[5] https://sourceforge.net/projects/ipt-netflow/
[6] http://www.ntop.org/products/netflow/nprobe/

Service, exportéry s více rozhraními přidávají identifikátor rozhraní[7]. Volba neklíčových elementů je závislá na uživateli a na možnostech exportéru, je ale jasné, že jejich množství a typ (především netriviálně extrahované elementy z aplikační vrstvy) mohou výrazně ovlivnit celkový výkon.

Dříve bylo popsáno, jak se mezipaměť toků plní a aktualizuje. Je ale nutné z ní záznamy také **mazat**, jinak by se tabulka plnila do nekonečna, nedocházelo by k expiraci záznamů a exportér by neplnil svou funkci. Záznam je v mezipaměti uchován od doby jeho vytvoření až do okamžiku, kdy je daný tok považován za ukončený. Pokyn k ukončení (expiraci) toku může přijít z několika zdrojů:

Aktivní časový limit: pakety příslušící do daného toku stále přicházejí na vstup, tok je tedy stále aktivní. Díky tomuto limitu je možné pozorovat dlouhotrvající toky průběžně, nikoliv jako jeden záznam vzniklý až po jeho ukončení. Jeho hodnota není pevně daná, Cisco zařízení umožňují nastavit rozsah od jedné sekundy po sedm dní, výchozí hodnota je třicet minut[8].

Pasivní časový limit: záznam v mezipaměti už nebyl aktualizovaný po dobu danou hodnotou tohoto limitu. Výchozí hodnota na Cisco zařízeních je 15 sekund.

Rozpoznání ukončení toku: u některých protokolů je možné rozpoznat ukončení toku díky detekci ukončení spojení. Jedná se třeba o protokol TCP a příznaky FIN nebo RST.

Jiné: zaplnění mezipaměti, manuální expirace vybraných nebo všech záznamů administrátorem.

Po expiraci lze provést druhé kolo vzorkování a filtrování, tentokrát však na úrovní toků. Principiálně se ale nijak neliší od stejného procesu na úrovni paketů popsaném v sekci 3.1. Zbývající záznamy jsou zapouzdřeny do aplikačního protokolu a odeslány na kolektor.

Formát zpráv u NetFlow v5 je jasný, protože informační elementy záznamu jsou pevně stanovené a tedy předem známé jak exportéru, tak kolektoru. U novějšího NetFlow v9 a z něj vycházejícího IPFIX je ale situace složitější, protože se exportér a kolektor musí domluvit na podobě záznamu. To se uskutečňuje prostřednictvím speciálních **šablonových zpráv** [15, 19], kterými exportér kolektoru sděluje které elementy a v jakém pořadí budou datové zprávy obsahovat. Šablonové i datové záznamy jsou označeny identifikačním číslem, je proto možné provádět současně export několika druhů záznamů. Kolektor potom zpracovává záznam na základě dříve doručené šablony s odpovídajícím identifikátorem.

Cílem aplikačních protokolů je také co nejlépe využít přenosovou linku, proto zprávy typicky obsahují tolik záznamů, aby se velikost IP paketu přiblížila MTU linky nebo celé trasy. Poměr režijních a užitečných dat se zlepšuje s rostoucím počtem záznamů v paketu. Posledním úkolem této fáze je odeslání paketu na exportér pomocí některého z **transportních protokolů**. Na výběr jsou dobře známé TCP a UDP, nabízí se ale také podstatně méně používaný Stream Control Transmission Protocol, SCTP. Tok dat mezi exportérem a kolektorem má proudový charakter stejně jako VoIP nebo přenos videa v reálném čase, kde se obvykle používá UDP. UDP je velmi jednoduchý, má malou režii, ale neposkytuje detekci ani korekci ztráty paketu tak stejně jako neposkytuje doručení v nezměněném pořadí [48].

[7] http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/flexible-netflow/prod_qas0900aecd804be091.html

[8] http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/fnetflow/command/fnf-cr-book/fnf-c1.html

Při ztrátě paketu obsahujícího datové záznamy jsou přenášené záznamy nenávratně ztraceny, mnohem větší ztrátu dat ale může způsobit ztráta paketu se šablonami. Bez nich kolektor nezná sémantiku datových záznamů a nemůže je tak zpracovávat. Nevýhodou je také chybějící kontrola zahlcení, která by musela být implementována aplikací. Většinu nedostatků UDP eliminuje použití protokolu TCP. Ten je spojovaný, garantuje spolehlivé doručování a doručování ve správném pořadí [60]. Protokol poskytuje kontrolu zahlcení prostřednictvím algoritmů jako slow-start, congestion avoidance, fast retransmit nebo fast recovery, které v případě detekce zahlcení způsobí (pomocí mechanizmu klouzavého okna) redukci množství přenášených dat. Problém nastává, když TCP redukuje rychlost přenosu dat natolik, že nestačí pro expirované záznamy o tocích. Exportér potom musí takové záznamy namísto exportu ukládat v paměti, což rozhodně není ideální. SCTP nabízí všechny zmiňované výhody TCP [55], navíc ale poskytuje rozšíření pro *částečnou spolehlivost*, které řeší problém zahlcení. Rozšíření definuje nový typ `FORWARD TSN` [56], kterým odesilatel (exportér) říká příjemci (kolektor), aby si posunul takzvaný *ack bod*. Tím dojde k přeskočení jedné či více datových částí, které zatím nemusely být přijaty a/nebo potvrzeny. Záznamy o tocích obsažené v přeskočených datech budou ztraceny, nedochází ale k zatěžování paměti exportéru jako v případě TCP.

3.3 Sběr a uložení dat

Vstupem pro fázi sběru dat jsou zapouzdřené záznamy o tocích od jednoho či více exportérů. Typický postup je příjem, zpracování a uložení dat. Příjem obnáší zpracování aplikačního protokolu: extrakce záznamů o tocích, správa šablon a mnoho dalšího. Zpracování záznamů je při ukládání méně obvyklé, je ale možné například generovat statistiky nebo vytvářet přehledové grafy (např. pomocí round-robin databáze, obrázek 2.1). Poslední a nejdůležitější bod je uložení dat, kde je možné volit mezi několika způsoby, formáty a kompresemi.

Nově příchozí toky je možné ukládat do operační paměti, která je rychlá a umožňuje tak efektivní analýzu dat. Má ale omezenou kapacitu, proto pokud chceme uchovávat data starší, než nám dovoluje kapacita operační paměti, je potřeba data přesunout na **perzistentní úložiště**. Tím se typicky rozumí pevné disky, které nabízejí podstatně větší kapacitu za cenu řádově nižší rychlosti zápisu i čtení. Je proto potřeba hledat vhodný databázový model, který bude poskytovat dostatečný výkon v oblasti vkládání i čtení dat. Další parametry, které při hledání hrají roli, jsou rychlost vyhledávání, flexibilita a síla dotazování, přenositelnost uložených dat nebo množství použitého místa na disku. Poměrně velký zájem o tuto oblast dokazuje řada článků [35, 34, 24, 61] vzniklých během posledních let, univerzálně nejlepší řešení ale výzkum zatím nepřinesl. To, co se při experimentech jeví jako nejlepší řešení, se totiž v praxi může ukázat jako problematické. Následující odstavce přinášejí přehled běžně používaných řešení.

Prostý databázový soubor, anglicky **flat file**, je fyzický datový model, který ukládá data bez hierarchie do obyčejného souboru [52]. Obsahuje pouze malé množství metadat a samotné záznamy jsou ukládány do souboru jeden za druhým (příchozí záznam je jednoduše připsán na konec souboru). Model si můžeme představit jako tabulku, kde řádky představují záznamy o tocích a sloupce jsou jejich informační elementy. Jsou úsporné co do množství uložených dat a poskytují vysoký ukládací výkon. Sekvenční čtení je sice rychlé, bohužel ale znamená potřebu přečíst všechny informační elementy každého záznamu (všechny sloupce v každém řádku). Pokud dotaz opravdu vyžaduje všechny, je to v pořádku. Pokud ale vyžaduje práci pouze s jejich podmnožinou, jsou generovány zbytečné V/V operace. Aby bylo možné dotazy směřovat jen na určité záznamy (na základě času jejich příchodu

ID	Pivovar	Název	Druh
1	Matuška	Raptor	IPA
2	ZP Frýdlant	Albrecht 12	Ležák
3	Falkon	Imperial	Stout

Tabulka 3.2: Vzorová logická tabulka pro znázornění rozdílu mezi různými přístupy fyzického uložení.

na kolektor), je vhodné provádět rotaci souborů, typicky se tak děje každých pět minut. Flexibilita dotazování závisí na aplikaci, která se soubory pracuje. Kolektory využívající tento způsob ukládání typicky neumí pracovat s dotazy ve formátu strukturovaného dotazovacího jazyku SQL, ale implementují vlastní způsob, nebo jazyk dotazování. Soubory mohou být jak binární, tak textové. U textových jsou informační elementy před uložením převedeny na odpovídající textovou reprezentaci. Jejich výhodou je dobrá přenositelnost, soubory lze jednoduše kopírovat, textová reprezentace navíc eliminuje problémy s rozdílnými datovými typy a endianitou. Převodem do textové reprezentace ale utrpí ukládací výkon, navíc při čtení je potřeba elementy opět převést do binární reprezentace. Také objem takových souborů je několikanásobně větší, což má za následek ještě větší množství V/V operací než u binárních.

K serializaci dat, tak aby mohly být zapsány na libovolné úložiště, se v oblasti databází využívají tři přístupy. Nejběžnější jsou *řádkově orientované databáze*, méně obvyklé jsou *sloupcově orientované databáze* [1] nebo *korelační databáze*. **Řádkově orientované databáze** pro uložení dat využívají prostých databázových souborů, což přináší všechny klady i zápory z předchozího odstavce. Tabulka 3.2 by mohla být uložena následovně:

```
1,Matuška,Raptor,IPA;2,ZP Frýdlant,Albrecht 12,Ležák;3,Falkon,Imperial,Stout
```

Hlavní nedostatek tohoto přístupu, nutnost čtení všech řádků při každém dotazu, se databázové systémy snaží eliminovat použitím indexů. Databázový index je kopie podmnožiny sloupců uložená nezávisle na primárním databázovém souboru, díky které je možné efektivněji číst a prohledávat pouze sloupce zahrnuté v dotazu. Cenou za indexy je vyšší počet zápisů při ukládání (některé sloupce je potřeba zapsat dvakrát) a samozřejmě také zabírání dodatečného prostoru v úložišti.

Principu indexů v řádkových databázích využívají **sloupcově orientované databáze**. Ty ukládají data po sloupcích, tabulka 3.2 by mohla být uložena následovně:

```
1,2,3;Matuška,ZP Frýdlant,Falkon;Raptor,Albrecht 12,Imperial;IPA,Ležák,Stout
```

Díky tomuto způsobu uložení může databázový systém přistupovat pouze ke sloupcům, které dotaz skutečně požaduje. Další výhodou je větší efektivita kompresních algoritmů [2, 1], protože sloupec obsahuje podobná data a lze jej tak v určitých případech komprimovat s lepším poměrem, než rozdílná data napříč řádkem. Lepší komprese způsobí snížení velikosti ukládaných souborů a tím pádem je při čtení ještě více redukován počet V/V operací. Použitím sloupcové databáze můžeme dosáhout výrazně kratších časů trvání dotazu [2], rozdíl lze pozorovat především na velkých datových sadách.

3.4 Analýza dat

Vstupními daty pro tuto závěrečnou fázi jsou záznamy o tocích, které byly uloženy do perzistentního datového úložiště. Existuje několik **důvodů**, proč je potřeba záznamy ana-

lyzovat. Prvním z nich je získání informací o tom, co se v síti děje. Ať už to jsou statistické pohledy na síť jako celek, nebo naopak pohledy cílené na konkrétní subjekty, oboje je možné z historických dat vyčíst. Druhý důvod je detekce útoků a hrozeb. Při analýze na kolektoru se obvykle zabýváme zamezení útoků, neboť doba od jeho uskutečnění až po případnou detekci je poměrně dlouhá (částečně ji však lze zkrátit proudovým zpracováním). Detekce útoků v historických datech může být prováděna manuálně (např. dohledávání komunikace konkrétních subjektů při forenzní analýze), nebo automatizovaně (uplatnění algoritmů pro detekci konkrétních typů útoků, hledání anomálií např. pomocí umělé inteligence [29]). Automatická detekce kromě správně detekovaných (a nedetekovaných) událostí vytváří také nežádoucí falešně pozitivní a negativní detekce, se kterými si musí poradit až administrátoři. Posledním zmiňovaným důvodem je monitorování výkonnosti sítě – jednotlivých prvků a služeb. Z toků lze zjistit parametry jako obousměrné zpoždění (round-trip delay time), jitter, aplikační doba odezvy a další.

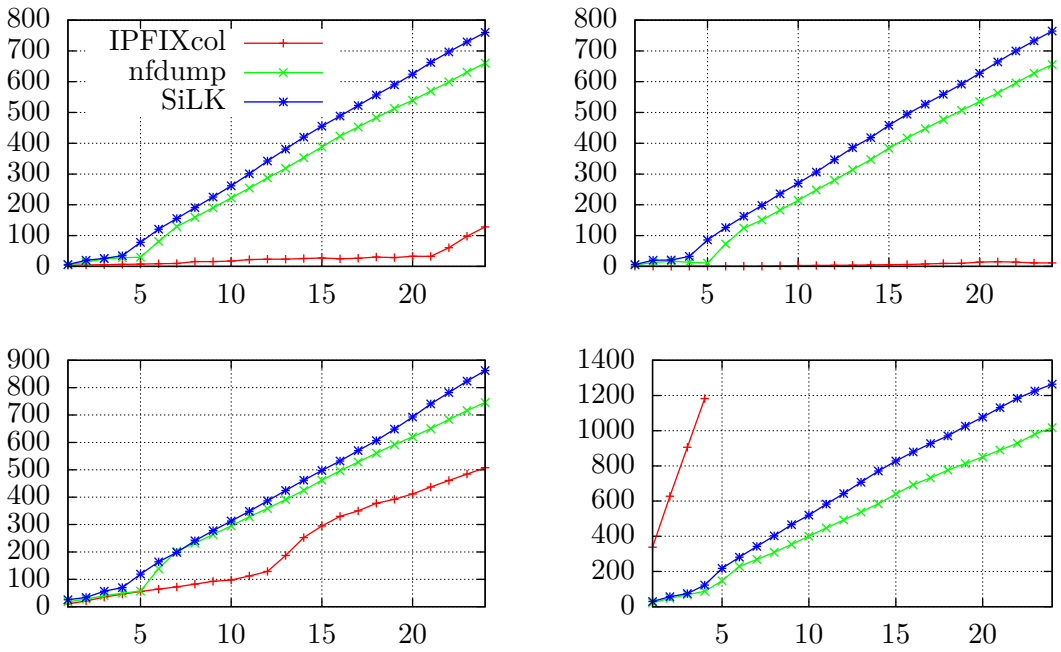
Jiná než proudová analýza obnáší práci s historickými záznamy o tocích, tedy čtení dat z perzistentního úložiště. Jak velké kvantum dat to bude, závisí na několika faktorech: velikost sledované sítě, objem provozu, nastavení vzorkování a filtrace, typ dotazu včetně toho, jak široký časový rozsah má zpracovávat. Monitorováním perimetru sítě CESNET2[9], kde je vzorkovací poměr 1:1 a data se ukládají do prostého databázového souboru ve formátu nfdump, vznikne denně více než 7 miliard záznamů o tocích, které po uložení zabírají zhruba 200 GB. Vetší sítě a ISP mohou samozřejmě produkovat ještě více dat a analytické dotazy, ať už manuální nebo automatizované, musí být dokončeny v rozumném čase. Tento požadavek ale kolektory fungující na centralizovaném systému nesplňují, při velkém množství dat je proto přesun na distribuovaný systém nevyhnutelný.

V předchozí sekci 3.3 byly zmíněny různé způsoby fyzického uložení dat v databázových systémech. Záznamy o tocích ale přestavují specifický případ použití těchto modelů, pojďme se podívat, jak se jejich obecné vlastnosti projevují na skutečných datech. Článek [34] porovnává relační řádkově orientovanou databázi MySQL[10] s nástrojem nfdump[11], který čte data z prostých databázových souborů. Testovací sada obsahuje 445 miliónů záznamů, velikost prostých souborů je 22 GB, MySQL databáze bez indexů zabírá 31 GB a s indexy velikost vzroste na 43 GB. Autor měří dobu odezvy u několika typů dotazů (výpis, výpis s filtrem, agregace s filtrem). Jasným vítězem srovnání se stal nástroj nfdump, který porazil databázi MySQL ve všech typech dotazů. Výsledné časy nástroje nfdump rostou lineárně spolu s rostoucím objemem dat, u MySQL s indexy křivka připomíná exponenciálu a u MySQL bez indexů je doba téměř konstantní.

Měření z [61] se zaměřuje na srovnání dotazovacího výkonu sloupcově orientované databáze a nástrojů nfdump a SiLK[12]. Sloupcová databáze je implementována jako součást kolektoru IPFIXcol[13], který využívá knihovny FastBit[14]. Datová sada obsahuje 1,1 miliard záznamů, velikost prostých souborů ve formátu nfdump i IPFIXcol je 53 GB, formát nástroje SiLK zabírá 67 GB. Způsob měření je obdobný jako v předchozím případě, stejná čtveřice dotazů je postupně spouštěna nad rostoucím množstvím dat:

```
Q1: SELECT count(*), sum(packets), sum(bytes) FROM dataset
Q2: SELECT count(*) FROM dataset WHERE dst port = 53
```

[9] https://www.cesnet.cz/sluzby/pripojeni/sit-cesnet2/
[10] https://www.mysql.com/
[11] http://nfdump.sourceforge.net/
[12] https://tools.netsa.cert.org/silk/
[13] https://github.com/CESNET/ipfixcol
[14] https://sdm.lbl.gov/fastbit/index.html



Obrázek 3.3: Výsledky měření z [61]. Horní dva grafy jsou dotazy bez agregace, spodní dva s agregací. Vodorovná osa znázorňuje růst datové sady po hodinách, svislá dobu běhu dotazu v sekundách.

```
Q3: SELECT src IPv4, packets, bytes, count(*) FROM dataset
    WHERE ip version = 4 GROUP BY src IPv4 ORDER BY bytes DESC LIMIT 5
Q4: SELECT protocol, src IPv4, dst IPv4, src IPv6, dst IPv6, src port,
    dst port, packets, bytes, count(*) FROM dataset GROUP BY protocol,
    src IPv4, dst IPv4, src IPv6, dst IPv6, src port, dst port
```

Na obrázku 3.3 vidíme, že při dotazech bez agregace vykazuje sloupcová databáze podstatně lepší dobu odezvy, která je spolu s rostoucím množstvím dat téměř konstantní. Tam, kde nfdump zpracovává data zhruba 10 minut, SiLK zhruba 12, IPFIXcolu to trvá jednotky vteřin. Při agregaci je rozdíl méně propastný, stále je ale IPFIXcol rychlejší přibližně o polovinu. Na jednom z dotazů, kde se agreguje na základě typické klíčové pětice (zdrojová a cílová IP adresa, zdrojový a cílový port, IP protokol), se projevil suboptimální agregační algoritmus knihovny FastBit. Odezva byla pomalejší a přílišná spotřeba paměti způsobila neúspěšné ukončení dotazu už při 20 % datové sady.

Velice podobné dotazy ve svém měření provádí také autor článku [24], který srovnává kolektor nProbe (využívající knihovnu FastBit) a relační řádkovou databázi MySQL. Datová sada je ale o poznání menší, obsahuje 68 miliónů záznamů, jejichž uložení zabírá v obou případech 2 GB dat bez indexů a přibližně dvojnásobek s indexy. Výsledky jsou srovnatelné s měřením z předchozího odstavce, sloupcová databáze poskytuje výsledky řádově rychleji. Zajímavým poznatkem je, že MySQL bez indexů ve všech dotazech poráží konfiguraci s indexy.

Kapitola 4

Vysoká dostupnost v distribuovaném systému

Kombinace vysoké dostupnosti a distribuovaných systémů vyžaduje znalost obou těchto pojmů, proto je v této kapitole nejprve rozebrán úvod do distribuovaných systémů, jejich cíle, klady, zápory nebo rozdíly oproti systémům centralizovaným. Následně je obecně popsána vysoká dostupnost, jsou také objasněny pojmy jako spolehlivost a provozuschopnost. Až potom je možné porozumět vysoké dostupnosti v distribuovaném systému, konkrétně v klastru počítačů.

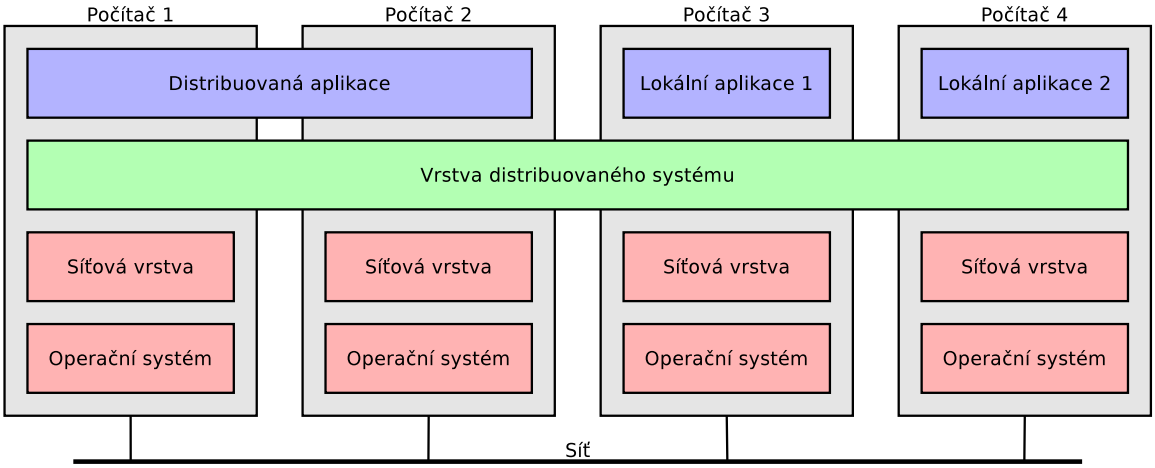
4.1 Distribuovaný systém

Internet, mobilní bezdrátové sítě nebo třeba DNS, to jsou široce používané distribuované systémy staré desítky let. V posledních letech se ale ukazuje, že distribuované systémy nacházejí uplatnění také v mnoha jiných odvětvích, krom zmiňovaných telekomunikačních a počítačových sítí. Významnou roli hrají v bankovnictví, elektronickém obchodování [6], leteckých a kosmických řídicích systémech, při paralelních výpočtech [22], v herním průmyslu nebo v oblasti elektronických peněz.

Definic distribuovaného systému (dále jen DS) je celá řada. [22] uvádí, že jde o systém, kde hardwarové a softwarové komponenty jsou umístěny na počítačové síti, vzájemně komunikují a koordinují akce pouze pomocí zasílání zpráv. Setkáváme se ale také s tvrzením, že jde o kolekci nezávislých počítačů, které na uživatele působí jako jeden logický systém [58]. Obecnější definice podle [39] zase říká, že jde o kolekci odlišných procesů, které jsou prostorově odděleny a vzájemně komunikují zasíláním zpráv. Všechny tyto mají jedno společné: jde o množinu procesů/počítačů komunikujících pomocí sítě. Můžeme také pozorovat první výrazné rozdíly oproti centralizovanému přístupu, kde jsou prostředky umístěny na jednom počítači, který je proto nezávislý na okolí a nemusí využívat žádné externí komunikace. Distribuovaná aplikace je potom taková, kde je programová logika rozprostřena mezi dvě nebo více komponent DS [6]. Příklad systému s lokální i distribuovanou aplikací je na obrázku 4.1.

4.1.1 Cíle

Vytvořit distribuovaný systém ale neznamená implicitní řešení všech problémů centralizovaného přístupu. Nemusí to být zkrátka vždycky výhodné, je proto dobré držet se následujících čtyřech cílů [58], které by měly být splněny, aby se vyplatilo DS vytvořit a udržovat:



Obrázek 4.1: Vrstva distribuovaného systému (middleware) se rozpíná přes jednotlivé počítače spojené sítí a poskytuje prostředí pro distribuované i lokální aplikace [58].

1. snadný přístup uživatelů a aplikací ke sdíleným prostředkům,
2. transparentnost,
3. otevřenost,
4. škálovatelnost.

Z ekonomického hlediska je je výhodné sdílet prostředky. Těmi mohou být tiskárny, soubory ale také třeba počítače a úložné prostory, téměř cokoliv. A právě **snadný přístup uživatelů a aplikací ke sdíleným prostředkům** je prvním z cílů DS. Takto dostupné zdroje mají navíc také efekt snadnější spolupráce a výměny informací mezi uživateli.

Dalším důležitým cílem je **transparentnost**. Její hlavní myšlenka je taková, že uživatel nemusí znát detaily architektury ani jednotlivé komponenty systému včetně jejich umístění, aby bylo možné na systém pohlížet jako na jeden logický celek. S distribuovaným systémem ale přijdou do styku také vývojáři aplikací, kteří už si podobnou abstrakci nemohou dovolit a potřebují znát vnitřní mechanizmy. Ne nezbytně všechny, ale především ty, které souvisejí s jejich prací. Organizace ISO proto zavedla následujících osm forem transparentnosti [37]: *Transparentnost přístupu* dovoluje použití stejných operací pro přístup na lokální i vzdálené prostředky. *Transparentnost lokace* umožňuje přístup ke zdrojům bez znalosti jejich skutečného nebo síťového umístění. *Transparentnost souběžnosti* dovoluje dvěma a více procesům přistupovat ke sdíleným zdrojům bez toho, aby se vzájemně ovlivnily. *Transparentnost replikace* dovoluje použití několika shodných zdrojů za účelem zvýšení spolehlivosti a výkonnosti. Uživatelé ani programátoři s nimi nemusí explicitně pracovat, potřebné procedury při výpadku nebo provádění vyvažování zátěže jsou transparentní. *Transparentnost selhání* skrývá chyby před aplikacemi a uživateli, kteří tak mohou pokračovat v práci bez ohledu na selhání softwarové nebo hardwarové komponenty. *Transparentnost mobility* umožňuje volný pohyb zdrojů v systémů bez vlivu na funkčnost. *Transparentnost výkonu* dovoluje rekonfiguraci s cílem zvýšení výkonu a konečně *transparentnost škálování* dovoluje rozšiřovat aplikace a systém bez vlivu na strukturu nebo aplikační algoritmy.

Neméně důležitá je také **otevřenost** systému. Otevřený systém je takový, který implementuje standardní protokoly a rozhraní. Zásadním bodem pro dosažení otevřenosti je veřejně dostupná specifikace a dokumentace komponent systému a rozhraní, díky kterým je

možné vzájemná komunikace procesů implementujících stejné rozhraní. Často zmiňovanou výhodou otevřeného systému je tak například nezávislost na dodavateli.

Posledním, ale neméně důležitým cílem je **škálovatelnost**. Škálovatelnost je schopnost systému zvládnout rostoucí objem práce, počet elementů a uživatelů a/nebo jeho potenciál k tomu být za tímto účelem rozšířen [12]. Škálovatelnost je měřitelná ve třech dimenzích. Za prvé, systém může být *velikostně škálovatelný*, což znamená, že můžeme jednoduše přidávat zdroje a uživatele. Za druhé, *zeměpisně škálovatelný* systém je takový, kde prostředky mohou být provozovány daleko od sebe. Třetí dimenzí je *administrativní škálovatelnost* která znamená, že systém může být jednoduše řízený i přes to, že se rozprostírá přes vícero organizací s různými administrativami a bezpečnostními politikami.

4.1.2 Výhody, nevýhody a problémy nutné řešit

Na začátku této sekce byl uveden příklad několika DS, které by jako centralizované pravděpodobně nedosáhly takové popularity. V čem je tedy síla distribuovaného systému? Jaké jsou **výhody** tohoto přístupu? Je jich hned několik, mezi ty základní patří [40, 58]:

Vysoký výkon: práci je možné rozdělit a využít tak paralelně více počítačů. Ty dohromady přináší vyšší výpočetní výkon a větší propustnost vstupně výstupních operací.

Sdílení prostředků: především těch, které nejsou jedním uživatelem či aplikací využity nepřetržitě.

Škálovatelnost: komponenty, především hardware, mohou být do systému inkrementálně přidávány.

Spolehlivost: redundance, existence více kopií prostředků, dat a služeb umožní úplné nahrazení vadné komponenty.

Zeměpisné umístění: díky komunikaci po síti není podmínkou distribuovaného systému umístění komponent na jednom místě.

Tyto výhody ale nejsou zcela implicitní, pro jejich dosažení je potřeba vhodný návrh jak systému, tak aplikace. Zároveň ale distribuovaný a paralelní systém více počítačů přináší řadu **problémů** a výzev, které je nutné řešit. Jedná se například o:

Heterogenitu: síť, hardware, operační systémy, programovací jazyky nebo rozdílné implementace stejné aplikace. Rozdílnosti v komponentách přinášejí potřebu specifikovat protokoly, které řeší vzájemnou kompatibilitu. Jako příklad uvedu protokol Ethernet, který zajišťuje vzájemnou kompatibilitu i přes rozdílné fyzické média. Stejné problémy je nutné řešit při použití heterogenního hardware: rozdílná endianita (pořadí bajtů ve slově) nebo rozdílné šířky datových typů. Ani různé operační systémy a použité programovací jazyky pravděpodobně nebudou kompatibilní. Rozdílné způsoby volání funkcí, správa paměti, přepínání procesů, to je pouze malá ukázka subsystémů, které mohou způsobovat potíže.

Bezpečnost: informační bezpečnost je tvořena třemi klíčovými body, kterými jsou důvěryhodnost, integrita a dostupnost. Zdroje umístěné do distribuovaného systému kladou stejné nároky na zmíněné podmínky jako tomu je u centralizovaného systému, jejich zajištění je ale obtížnější. Mohou být rozmístěny přes více počítačů umístěných

třeba na jiných kontinentech. Ty potřebují vzájemně komunikovat a vyměňovat potencionálně citlivá data. Pokud není komunikace zajištěna vyhrazenou linkou, vzniká bezpečnostní riziko. Při komunikaci klientů se systémem je situace obdobná.

Kvalitu služeb: ta je ovlivněna výkonností, spolehlivostí a bezpečností. O jak vysokou kvalitu a jaké služby se jedná záleží na aplikaci. Ty, které operují s časově kritickými daty bezesporu potřebují zajistit dostatečný, stabilní výkon a propustnost pro jejich zpracování v určeném (třeba i reálném) čase. Jedná se o stejný princip jaký funguje v počítačových sítích, kde určitý provoz musí být odbavován přednostně (IP telefonie atp.).

Management: více komponent znamená větší nároky na správu. To s sebou přináší větší pravděpodobnost vzniku chyb v konfiguraci.

4.2 Vysoká dostupnost obecně

Obecně chápeme dostupnost jako vlastnost zdroje, která říká, zda je přístupný nebo použitelný. Tato vlastnost je měřitelná, proto před ní (obzvláště v IT) často přidáváme přídavné jméno, které míru určuje. Vysokou dostupnost lze **intuitivně chápat** tak, že systém byl navržen, implementován a je provozován v souladu s ochranou před nežádoucími výpadky [50]. Každá ztráta služby, ať už plánovaná nebo nikoliv, je nazývaná výpadek. Doba, po kterou je služba nedostupná se nazývá *prostoj* (*downtime*) a měří se v jednotkách času. Proč je potřeba chránit systémy a jejich zdroje před výpadky? Ve světě komerce znamenají výpadky finanční ztráty, odhady říkají, že v roce 1996 díky výpadkům informačních technologií došlo u Amerických podniků ke ztrátám v hodnotě miliard dolarů [36]. Když se nejedná o peníze, může být negativním dopadem třeba ztráta dat nebo špatná reputace služby.

Pokud se ale budeme snažit **definovat** vysokou dostupnost jinak než intuitivně, narazíme na problém. Je to totiž jeden z pojmů, kterému každý rozumí, ale neexistuje žádná jednotná precizní definice. Ve [59] je definována jako „charakteristika systému určená k ochraně před minoritními výpadky a obnově z nich, v krátkém časovém intervalu a zároveň je z velké části automatizovaná“. Jinde se uvádí, že v oboru informatiky se jedná o systém nepřetržitě provozuschopný a připravený poskytovat služby koncovým uživatelům [50].

Je potřeba si ale dát pozor na to, aby mechanizmy používané pro zajištění vysoké dostupnosti **nebyly příliš komplikované**. Můžou se tak stát kontraproduktivní a mít negativní dopad na spolehlivost, výkonnost nebo na samotnou dostupnost systému [51]. Proto je před návrhem spolehlivého systému důležité určit si cíle, kterých chceme dosáhnout. Nejsou totiž vymezeny hranice, za kterými bychom již mohli hovořit o vysoké dostupnosti, nicméně obvykle jde o 99,9 % a více času, kdy je systém dostupný (měřeno ročně). Abychom si ujasnili požadavky, je důležité znát odpovědi na následující otázky [51]:

1. Jaká selhání by měla být transparentní? Kde nesmí nesmí dojít k výpadku? V takových místech musíme implementovat *ochranu proti poruchám*, což je metoda která zajišťuje nepřetržitou dostupnost. Nepřetržitá dostupnost je drahá, proto je vhodné ji nasadit pouze u komponent, kde je to nezbytně nutné nebo u nich dochází k chybám často (pevné disky, napájecí zdroje).

2. Jak dlouho může trvat častější *minoritní výpadek*? Je potřeba se zamyslet nad tím, jak dlouho mohou být aplikace/uživatelé bez poskytovaných služeb, než dojde k vážným ztrátám. Častější krátký výpadek nemusí nutně znamenat velký problém a proto by bylo zbytečné zabývat se vyšší mírou spolehlivosti, než je nutné.

Dostupnost	Třída	Roční výpadek	Denní výpadek
90 %	Jedna devítka	36,5 dnů	2,4 hodiny
99 %	Dvě devítky	3,65 dnů	14,4 minuty
99,9 %	Tři devítky	8,76 hodin	1,44 minuty
99,99 %	Čtyři devítky	52,56 minut	8,66 sekund
99,999 %	Pět devítek	5,26 minut	864,3 milisekundy

Tabulka 4.1: Tabulka zobrazuje maximální dobu výpadku při dané třídě dostupnosti.

3. Jak dlouho může trvat méně častý *majoritní výpadek*? Majoritní výpadek je možné nazvat také pohromou nebo katastrofou a zpravidla se jedná o poruchu, při které dochází k závažným škodám na systému. Může jít o živelnou pohromu, požár, chybu administrátora nebo třeba úmysl.

4. O jaké množství dat můžeme přijít při majoritním výpadku? Která data jsou nenávratně ztracena a která lze znovu získat?

5. Která selhání jsou považována za tak nepravděpodobná, že se nemusí vůbec řešit?

Dostupnost, spolehlivost a provozuschopnost je trojice vlastností, pomocí kterých je možné vyjádřit **kvalitu systému**. Dohromady se označují zkratkou RAS z anglických výrazů reliability (spolehlivost), availability (dostupnost) a serviceability (provozuschopnost). Následující odstavce uvádějí jejich definice a způsoby výpočtu.

Dostupnost je míra, která určuje jak často a jak dlouho je služba/systém/komponenta dostupná k použití. Lze ji ale také vyjádřit jako pravděpodobnost s jakou služba/systém/-komponenta vykonává plánovanou činnost [59]. Míru dostupnosti lze vypočítat jako poměr doby, kdy byl systém dostupný a celkové doby

$$dostupnost = \frac{doba\ dostupnosti}{doba\ dostupnosti + doba\ nedostupnosti}. \tag{4.1}$$

Může být vyjádřena zlomkem, ale běžnější je procentuální dostupnost, tak jak je zapsána v tabulce 4.1. Přesné doby ale známe až posléze. Pokud máme dostupné hodnoty *střední doby mezi poruchami* (*Mean Time Between Failures*, MTBF) a *střední doby do obnovení* (*Mean Time To Repair*, MTTR), lze dostupnost odhadnout s předstihem jako

$$dostupnost = \frac{MTBF}{MTBF + MTTR}. \tag{4.2}$$

MTBF představuje předpokládaný čas mezi po sobě jdoucími poruchami, MTTR je čas, po který trvá oprava systému po nastalé poruše. Takto vypočtená dostupnost bude ale vždy pouhý odhad, protože samotné atributy MTBF a MTTR nelze přesně předpovědět, pouze odhadnout [59].

Spolehlivost je míra vyvarování se chyb, tedy pravděpodobnost s jakou bude systém v operativním stavu v daném časovém intervalu T

$$R(t) = P(T > t) = 1 - F(t), \tag{4.3}$$

kde $F(t)$ je funkce nespolehlivosti [51]. Je důležité si ale uvědomit, že se pohybujeme na poli matematické statistiky a pro zisk přesného výsledku potřebujeme statisticky významný vzorek.

Provozuschopnost je míra, která vyjadřuje, jak snadné je systém provozovat nebo opravovat. Jinak řečeno jde o to, kolik práce a času je potřeba do systému vložit k jeho znovuzprovoznění po poruše. Při návrhu systému je možné určit tzv. plánovanou obsluhovatelnost (např. pět hodin za měsíc). Tato plánovaná obsluha systému s sebou může přinést plánovaný výpadek, nevylučuje se ale možnost provedení zásahu za běhu. Naproti tomu skutečnou, neplánovatelnou obsluhovatelnost ale při návrhu určit nelze.

4.3 Vysoká dostupnost v klastru počítačů

Předchozí sekce 4.1 se zabývala distribuovanými systémy, zmíněny byly jejich cíle, výhody a nevýhody. V sekci 4.2 byly rozebrány vlastnosti systémů s vysokou dostupností bez zaměření na komponentu, počítač či více počítačů. Tato sekce je již konkrétnější a zabývá se vysokou dostupností aplikovanou na množinu dvou a více počítačů, která bude dále nazývána jako klastr.

Základním pravidlem pro vysoce dostupný systém je **eliminace jediných bodů selhání** (single point of failure, dále jen SPOF) [7]. Jde o komponenty, které svým selháním způsobí výpadek. Aby bylo možné zajistit vysokou dostupnost v centralizovaném systému, je potřeba specializovaný hardware, komponenty podporující výměnu vadného kusu za běhu (how swapping), redundanci na několika vrstvách atp. SPOF je v počítači celá řada, takže složitost i cena takového řešení se úměrně zvyšuje spolu s požadovanou třídou dostupnosti. Existují ale také softwarové SPOF v podobě aplikací poskytující služby, jejichž odstranění je v centralizovaném systému ještě složitější.

Použití klastru má v tomto ohledu značnou výhodu: při selhání počítače stačí, aby jeho funkci převzal jiný počítač. Oba mohou být složeny z komoditního[1] hardwaru bez redundantních komponent. Tento mechanizmus zvaný **failover** je ale nutné zajistit programově.

Existují **dva typy počítačových klastrů**: pro vyvažování zátěže a pro vysokou dostupnost [50]. První zmiňovaný se používá tam, kde je potřeba vysoký výkon pro obsluhu mnoha požadavků, pro vědecké výpočty atd. Důležitá je funkce, pomocí které je práce mezi uzly klastru rozdělována. Nejjednodušší možnost, kde je N-tý požadavek přidělen K-tému stroji podle rovnice $K = N \bmod K_{total}$, nebude vždy fungovat. Důvodem může být potřeba udržovat stav, který spolu uzly nesdílí nebo rozdílná náročnost jednotlivých požadavků. Existují proto sofistikovanější funkce pro rozdělování práce [32], ty ale nejsou předmětem tohoto textu. Druhý typ klastrů se primárně zaměřuje na vysokou dostupnost, výkon který je od něj požadován můžeme dosáhnout i na jednom uzlu. Velice užitečné je ale tyto přístupy kombinovat. Různé **možnosti konfigurace** popisuje následující výčet [50]:

Aktivní/aktivní: služba poskytovaná klastrem je současně aktivní na všech uzlech. Každý uzel je ekvivalentně připraven obsloužit požadavek. Pokud je implementováno vyvažování zátěže, požadavky by měly být distribuovány rovnoměrně mezi všechny uzly. Při poruše některého z uzlů na něj vyvažovač přestane směrovat požadavky ihned po detekci poruchy. Bez vyvažování zátěže jsou všechny požadavky obsluhovány jedním uzlem, všechny ostatní čekají na jeho poruchu a jsou připraveny bezprostředně převzít jeho funkci při výpadku. Tato konfigurace poskytuje nejrychlejší čas konvergence, její použití je ale možné pouze v případě homogenní softwarové konfigurace všech uzlů a pro její uplatnění je potřeba dva a více uzlů.

Aktivní/pasivní: služba poskytována klastrem je aktivní pouze na jednom uzlu. Všechny

[1] široce rozšířený, snadno dostupný a levný

ostatní uzly jsou pasivní a při poruše aktivního uzlu dojde k automatické rekonfiguraci klastru. Jeden z pasivních uzlů je zvolen jako nový aktivní uzel a přebírá veškerou jeho zodpovědnost. Konvergenční čas závisí na trvání detekce poruchy aktivního uzlu a následné rekonfiguraci, typicky je horší než u konfigurace aktivní/aktivní. Stejně jako v případě předchozí konfigurace je potřeba dva a více uzlů.

N + 1: v klastru je aktivních N uzlů s možnou heterogenní softwarovou konfigurací (každý může poskytovat jinou službu). Jeden uzel je pasivní a při detekci poruchy libovolného aktivního uzlu převezme jeho roli. Při heterogenní konfiguraci musí být připraven obsloužit libovolnou ze služeb poskytovaných klastrem. Existují dvě varianty konfigurace, které ovlivní chování systému po opravě porouchaného uzlu. Při první z nich se opravený uzel nevrací do své původní role, ale převezme roli náhradního uzlu čekajícího na poruchu. Při druhé variantě se opravený uzel stává zase aktivním a uzel, který jej dočasně zastoupil se stává zase pasivním (tato varianta se někdy označuje jako preemptivní nebo N-to-1). Druhá zmiňovaná varianta je vhodná v případě, kdy je jeden uzel výkonnější a je tak žádoucí, aby byl aktivní vždy když je to možné. Pro tuto konfiguraci je už ale potřeba více než dva uzly.

N + M: služba poskytovaná klastrem je současně aktivní na N uzlech. Zbývajících M uzlů je pasivních a platí pro ně stejná pravidla jako pro pasivní uzel v konfiguraci N + 1. Nabízejí se stejné dvě varianty konfigurace (preemptivní a nepreemptivní) a také je také potřeba více než dva uzly.

4.4 Další pojmy z oblasti distribuovaných systémů

Následující sekce příliš nenavazují na předchozí text, jde však o pojmy z oblasti distribuovaných systémů, které jsou používány v implementační části práce a je potřeba je objasnit.

4.4.1 Split-brain, rozdělení sítě a kvórum

Pojem **split-brain** se používá jako analogie k syndromu oddělených hemisfér, který se v angličtině nazývá též split-brain. Jde o (typicky nežádoucí) stav, kdy několik disjunktních množin uzlů obsluhují sdílené prostředky, přičemž množiny o sobě nevědí a nekomunikují spolu [44]. Split-brain vede k porušení konzistence na úrovni služeb nebo dat. Typicky je totiž sdíleným prostředkem úložiště, kde souběžný přístup několika nekomunikujících uzlů může způsobit poškození uložených dat.

Disjunktní nekomunikující množiny uzlů (tedy vlastně několik nezávislých klastrů) mohou vzniknout dvěma způsoby: špatným nastavením a selháním síťové komponenty. Druhý zmiňovaný případ se nazývá **rozdělení sítě**, vzniklé množiny se nazývají *komponenty* [22], stejně jako v teorii grafů. Jako příklad uvedu Ethernetový přepínač, pomocí nějž uzly komunikují a ověřují stav ostatních uzlů. V případě jeho selhání uzly zůstanou funkční, každý z nich si ale bude myslet, že všechny ostatní uzly v klastru selhaly. Na základě této mylné informace uzel myslí, že získal exkluzivní přístup ke sdíleným prostředkům a to může vést k operacím vedoucím k nekonzistenci a poškození dat.

Jak ale praví **CAP teorém** [13], DS nemůže poskytovat zároveň konzistenci, dostupnost a odolnost vůči rozdělení sítě. Je možné garantovat pouze dvě ze zmiňovaných vlastností a jednu „obětovat“. V případě distribuovaného kolektoru se chceme vypořádat s rozdělením sítě na několik komponent a zároveň nepřipustit žádné inkonzistence v datech. Je tedy nutné

obětovat dostupnost tak, že v případě rozdělení sítě povolíme provádět operace pouze jediné komponentě, ostatní budou izolovány.

Jednou z metod pro výběr vhodné aktivní komponenty, je **metoda kvóra** [22]. Slovník cizích slov definuje kvórum jako „nejnižší počet členů nějakého orgánu, aby byl usnášeníschopný“ nebo jako „nejnižší počet hlasů předepsaný jako podmínka pro získání mandátu“ [3]. Tento politický pojem má podobný význam i na poli DS: jde o nejnižší počet hlasů, který musí být získán, aby bylo možné v DS provádět operace. Vhodnou hodnotou kvóra K pro klastr o počtu uzlů $N > 2$ je

$$K = \left\lfloor \frac{N}{2} \right\rfloor + 1. \tag{4.4}$$

Je potřeba získat nadpoloviční počet hlasů, který může získat vždy pouze jediná komponenta (ta s největším počtem přítomných uzlů). V případě, že žádná z komponent nezíská nadpoloviční většinu hlasů (viz příklad se selháním přepínače), celý klastr se sice stane nedostupný, ale nedojde ke vzniku nekonzistence.

4.4.2 Fencing a STONITH

Fencing je proces neplánovaného odstranění uzlu z klastru a jeho izolace od všech sdílených zdrojů [33]. Uzel A může takto izolovat sám sebe, např. z důvodu ztráty kvóra, nebo může být izolován jiným uzlem B po zjištění, že na uzlu A došlo k poruše. Důvodem této akce je ochrana sdílených prostředků, porouchaný uzel by k nim totiž mohl přistupovat nesprávným způsobem a vytvořit tak nekonzistence nebo poškodit data.

Izolace uzlu je pouze virtuální, žádné fyzické odpojování kabelů není potřeba. Stačí totiž zajistit, že na uzlu nebudou běžet žádné služby. Ve chvíli, kdy se lze spolehnout na operační systém (uzel nemá poruchu, tj. situace při ztrátě kvóra), zajistí uzel ukončení všech služeb sám. V případě poruchy uzlu ale nemusí OS vůbec odpovídat, v tom případě se o izolaci musí postarat některý z funkčních uzlů klastru pomocí metody zvané **STONITH** (Shoot The Other Node In The Head). Tato metoda zahrnuje násilné restartování nebo vypnutí uzlu a to buď odpojením od přívodu elektrické energie (PDU v racku), nebo pomocí systému nezávislé správy počítače (IPMI) [50]. Tímto proces automatické izolace končí a další kroky jsou na administrátorovi, který musí zjistit příčinu izolace uzlu a zajistit nápravu.

4.4.3 Virtuální synchronie

Některé aplikace mohou v distribuovaném systému fungovat stejně jako v centralizovaném, tzn. nemusí spolu přímo spolupracovat, jakkoliv sdílet stav nebo si vyměňovat zprávy. V opačném případě mluvíme o aplikacích, které jsou vytvořeny nebo uzpůsobeny pro fungování v distribuovaném systému. Spolu s výhodami, které takové aplikace přináší ale přicházejí také nové **problémy** [10]:

- souběžnost, nejistota stavu ostatních procesů,
- detekce výpadků, dynamická rekonfigurace, obnova po výpadku,
- konzistence
- rozdělení procesů na skupiny, komunikace pouze v rámci skupiny.

Kvůli absenci sdílené paměti spočívá jediný způsob vzájemné komunikace procesů ve výměně zpráv. V synchronním prostředí, kde je multicast mezi procesy atomický a události (výpadky, obnovy, ...) nastávají všude ve stejném pořadí, jsou procesy vždy implicitně ve stejném stavu a udržovat konzistentní stav systému tak není problém. Konvenční komunikační sítě ale neposkytují požadované vlastnosti a implementace protokolu zajišťujícího synchronnost by byla neefektivní [10]. Čas a pořadí přenesení zpráv jednotlivým procesům jsou závislé na mnoha faktorech, obecně ale musíme předpokládat, že zprávy mohu být k procesům přeneseny za rozdílnou dobu v rozdílném pořadí nebo nemusí být doručeny vůbec. Výpadek je možné detekovat až po uplynutí předem určeného časového limitu, takže po dobu mezi výpadkem a uplynutím limitu není možné rozeznat ztrátu zprávy od výpadku procesu. Tyto faktory komplikují udržování konzistentního pohledu procesů na stav ostatních, koordinace procesů se stává náchylná k chybám a neefektivní.

Pro mnoho aplikací jsou ale atomický multicast a úplné uspořádání zbytečně silné, vznikl tak koncept **virtuální synchronie** [10, 11]. Ten je vhodný v situaci, kdy aplikace není citlivá na pořadí doručení, např. když jsou zprávy doručovány v pořadí, v jakém je proces odeslal ale jsou neuspořádané se zprávami z jiných zdrojů. Jakémukoliv procesu se v prostředí virtuální synchronie bude jevit, že všechny procesy vidí veškeré události ve stejném pořadí. Událostmi se myslí doručení zprávy, selhání a obnova procesů, dobrovolné změny členství ve skupině a další. Každý proces tak může vytvářet předpoklady o stavu ostatních procesů, což s ohledem na podmínku globální korektnosti značně zjednoduší návrh ale i implementaci aplikace. Pokud dojde k rozdělení sítě (viz sekce 4.4.1), virtuální synchronie také zajišťuje, že procesům je umožněno pokračovat ve vykonávání programu pouze v jedné (hlavní) komponentě. Procesy v ostatních komponentách jsou zablokovány. Členové komponenty spolu s unikátním identifikátorem tvoří *konfiguraci*. Jeden z klíčových principů virtuální synchronie je postaven na rozlišování mezi *příjmem* zprávy přes komunikační médium a *doručením* zprávy cílové aplikaci. Příjem není možné ovlivnit, doručení ovšem ano a to jak z pohledu času tak pořadí. Rozlišujeme tři způsoby doručení zprávy: kauzální, dohodnuté a bezpečné. Každý způsob poskytuje jiné záruky [46], kauzální doručení se vztahuje pouze na zprávy ve stejné konfiguraci, dohodnuté a bezpečné doručení zaručují doručení zpráv v dohodnutém pořadí v konfiguraci, která předchází zprávě uvozující změnu konfigurace. Bezpečné doručení navíc garantuje, že pokud je zpráva doručena nějakým procesem, bude doručena každým dalším procesem v konfiguraci (pokud proces neselže).

Rozdělením a sloučením komponent sítě nebo selháním a obnovou procesů se stabilním úložištěm (stable storage) mohou v modelu virtuální synchronie nastat nekonzistence. Proto byl navržen model **rozšířené virtuální synchronie** [46], který ji rozšiřuje především v oblasti rozdělování a slučování komponent sítě. Model přidává druhý typ konfigurace, *přechodnou konfiguraci*. V běžné konfiguraci jsou nové zprávy vysílány a doručovány, v přechodné konfiguraci jsou pouze doručovány zprávy z předcházející běžné konfigurace. Žádné nové zprávy v přechodné konfiguraci nejsou vysílány.

Kapitola 5

Návrh rozšíření distribuovaného kolektoru

V této kapitole jsou rozebrány různé techniky a přístupy, kterými můžeme dosáhnout požadovaných cílů. Aby ale bylo možné vůbec něco upravovat, je potřeba znát současný stav, ve kterém se distribuovaný kolektor nachází. Dále je v kapitole popsán návrh dvou nových architektur distribuovaného kolektoru, které ze současného stavu vychází a vnáší do něj prvky zajišťující vysokou dostupnost a spolehlivost. Vysoká dostupnost a spolehlivost je klíčová, není ale jediným cílem práce. Při návrhu rozšíření jsem se zaměřil také na cíle obecného distribuovaného systému, kterými jsou snadný přístup uživatelů ke sdíleným prostředkům, transparentnost, otevřenost a škálovatelnost.

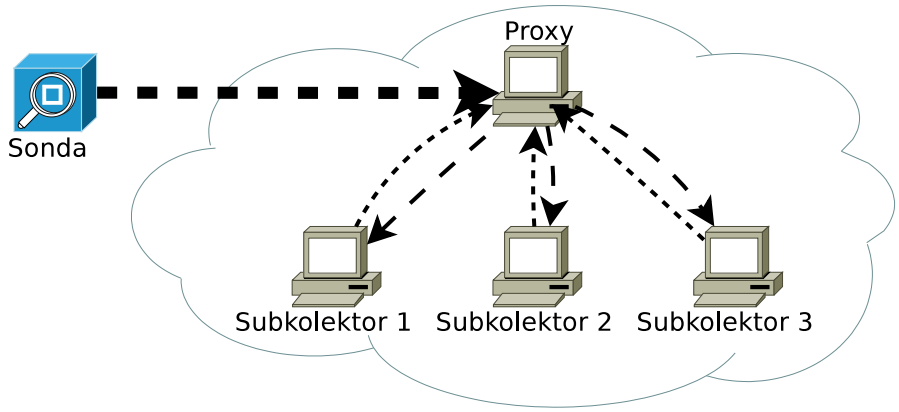
5.1 Současný stav

V současné době je de facto standardem použití **centralizovaného systému** pro kolektor záznamů o IP tocích. Jeho výhody a nevýhody zhruba odpovídají obecnému centralizovanému systému, jak je zmíněn v sekci 4.1. Nabízí jednoduchou správu, existuje několik volně dostupných i komerčních řešení. Nevýhody jsou především špatná škálovatelnost výkonu a problematické zajištění vysoké dostupnosti. Rychlost ukládání při velkém množství generovaných záznamů (např. při DDoS útocích) nemusí stačit, stejně tak je omezen výkon analytické části, tedy dotazování.

Problémy centralizovaného systému vedly ke vzniku konceptu nasazení kolektoru na **distribuovaný systém** [69]. Jeho architektura je zobrazena na obrázku 5.1 a tvoří ji dva druhy uzlů: *proxy*[1] a *subkolektor*. Zjednodušeně by se dal proxy uzel nazvat jako řídící a subkolektor jako pracovní, funkce obou uzlů je rozdílná z pohledu ukládacího a dotazovacího procesu:

Ukládání: primárním úkolem proxy uzlu je rozdělovat záznamy o tocích přicházející ze sondy na subkolektory. V současnosti je jediným možným způsobem rozdělení rovnoměrně algoritmem round-robin. Subkolektor potom tyto záznamy přijímá (úplně stejně jako přijímá standardní kolektory záznamy ze sondy) a ukládá na lokální úložiště. Proxy uzel je právě jeden, subkolektor musí být alespoň jeden. Obě funkcionality jsou programově zajištěny nástrojem IPFIXcol. Na proxy uzlu je spuštěn ve speciálním módu rozdělování záznamů a pro subkolektory se jeví jako exportér. Na všech

[1] Proxy uzel je typ prvku, který se v oblasti DS běžně označuje jako vyvažovač zátěže (load balancer). Název proxy je pro něj ale už zavedený, a proto jej budu používat i v této práci.



Obrázek 5.1: Současná architektura distribuovaného kolektoru. Čárkované čáry představují záznamy o tocích, tečkované jsou částečné výsledky dotazu.

subkolektorech potom běží klasická instance IPFIXcolu, která je nezávislá na ostatních.

Dotazování: dotaz typicky musí operovat se záznamy uloženými během určitého časového intervalu. Tyto záznamy jsou ale díky ukládací fázi rovnoměrně rozprostřeny mezi subkolektory. Prvním krokem každého dotazu je proto spuštění poddotazu na všech subkolektorech tak, aby sjednocení množiny zpracovaných záznamů bylo ekvivalentní původní množině před rozdělením proxy uzlem. Každý subkolektor tak nezávisle dospěje k nějakému částečnému výsledku. Proxy uzel provede jejich sběr, následnou agregací částečných výsledků vznikne konečný výsledek na dotaz. Tuto funkcionalitu implementuje nástroj fdistdump.

Dva stejné logické prvky nemohou sdílet jeden počítač, dva rozdílné naopak mohou. Je možné, aby proxy uzel a subkolektor sdíleli jeden počítač, dva subkolektory ale musí být na dvou počítačích. Minimální počet fyzických uzlů v klastru jsou dva: oba dva v roli subkolektoru a jeden navíc v roli proxy. Maximální počet není nijak omezen, množinu subkolektorů je možné neomezeně rozšiřovat.

Současný stav ale **není ideální** především z pohledu vysoké dostupnosti, která se v architektuře vůbec neřeší. Oba dva prvky architektury jsou SPOF a při každé poruše dochází k trvalé ztrátě dat. V případě proxy uzlu jde o data příchozí, všechny záznamy od exportérů budou zahazovány. Ani dotazování nebude fungovat. V případě dočasné poruchy subkolektoru nebude uložena část příchozích záznamů, v případě nevratné poruchy jeho lokálního úložiště budou trvale ztracena veškerá data uložená na subkolektoru. Nedostupnost jednoho subkolektoru neznemožní provádět dotazy, výsledky ale nebudou korektní, protože část dat bude chybět. Oproti centralizovanému systému tak je spolehlivost dokonce nižší, protože komponenty v systému přibyly.

5.2 Návrh nové architektury

V této sekci jsou představeny dvě architektury, které vycházejí ze současného stavu. Obě mají své klady i zápory, shrnutí je v tabulce 5.1.

Vlastnost/Architektura	Současná	Č. 1	Č. 2
Náklady	0	-	0
Odolnost proti minoritním poruchám	-	+	+
Odolnost proti majoritním poruchám	-	+	-
Množství hardware	+	-	+
Náročnost implementace	+	0	-
Dotazovací výkon	0	+	0

Tabulka 5.1: Přehled výhod a nevýhod všech architektur. Plus značí přednost v dané oblasti, mínus slabinu, nula je neutrální hodnocení.

5.2.1 Architektura 1: duplikace klastru

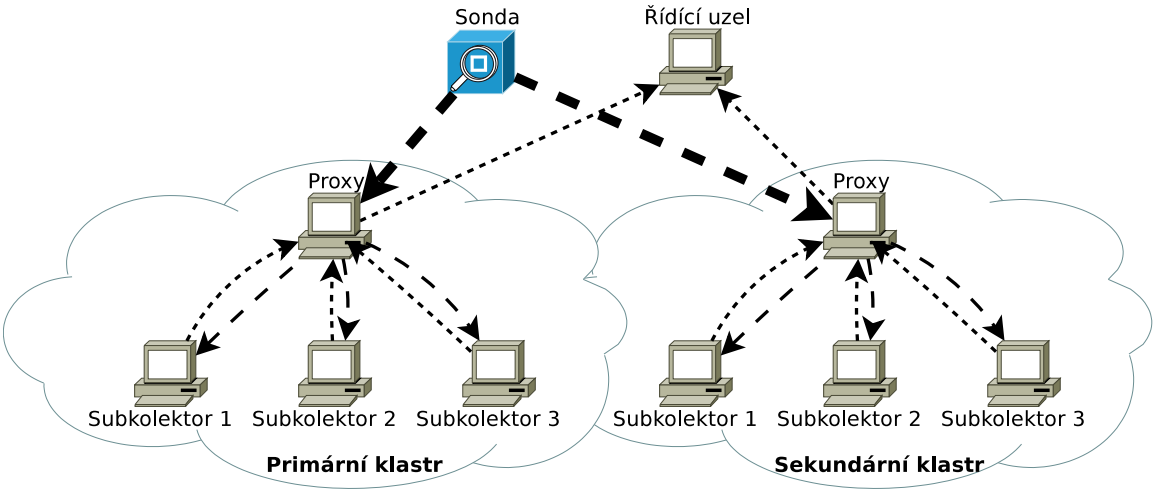
Jednoduchá eliminace SPOF u centralizovaného systému spočívá ve vytvoření jeho kopie (připadne kopií), která v případě poruchy primárního systému zastane jeho roli. Stejný princip můžeme využít také u distribuovaného systému, stačí přidat do současné architektury ještě jeden naprosto identický klastr (obrázek 5.2).

Exportér musí posílat **záznamy ve dvou kopiích**, jednu na proxy uzel každého klastru. Fáze pro ukládání dat se tak bude vykonávat na obou klastrech současně a záznamy o tocích tak budou uloženy ve dvou kopiích. Pro každý klastr zvlášť i nadále platí všechny jediné body selhání (každá porucha znamená ztrátu dat a znemožní podávat korektní odpovědi na dotazy). Díky duplikaci si ale můžeme dovolit označit klastr, kde k výpadku došlo, za porouchaný, a po dobu poruchy směřovat dotazy na ten druhý. Po opravě poruchy je ale potřeba uvést oba klastry do konzistentního stavu, na opraveném klastru totiž bude chybět část dat. Záložní klastr ale všechna data z exportérů ukládal, proto je možné provést jejich úplnou obnovu a navrátit oba subsystémy do konzistentního stavu.

Duplikací klastru bychom ale přišli o **centrální prvek**, kterým je v současné architektuře proxy. Řešením je přidat do architektury řídící uzel. Ten slouží jako přístupový bod, běží na něm služby obstarávající management, zároveň se ale stará o kontrolu stavu primárního i sekundárního klastru. V případě zjištění poruchy primárního klastru (ať už proxy uzlu nebo některého subkolektoru) totiž může transparentně směřovat všechny dotazy na sekundární klastr. Po obnově plné funkčnosti spustí řídící uzel proces synchronizace, který obnoví datovou konzistenci mezi oběma subsystémy.

Výsledkem je eliminace obou SPOF současné architektury. Oba klastry mohou být umístěny v datových centrech na jiných zeměpisných lokalitách aniž by to negativně ovlivnilo výkon. To s sebou přináší také ochranu proti majoritním poruchám a katastrofám, které způsobí poruchu/zničení více uzlů jednoho klastru současně. Další **výhodou** je dvojnásobný výkon analytické části, je totiž možné spustit souběžně dva dotazy, každý na jiném klastru. Uložená data jsou konzistentní, proto dva shodné dotazy položené ve stejnou dobu musí vykazovat tentýž výsledek na obou klastrech. Novým SPOF v této architektuře je řídící uzel, který lze ale jednoduše eliminovat duplikací. Tato architektura tedy přináší relativně snadnou eliminaci všech SPOF a dvojnásobný dotazovací výkon.

Její **nevýhodou** jsou vysoké pořizovací náklady a náročnost následné správy a údržby. Duplikací proxy uzlů, všech subkolektorů a přidáním dvou řídících uzlů totiž vzniká potřeba více než dvojnásobku uzlů, než je potřeba v současné architektuře. Dále export záznamů ve dvou kopiích způsobí dodatečnou zátěž exportéru a linky, kterou je připojen do sítě.



Obrázek 5.2: Architektura 1: duplikace klastru.

5.2.2 Architektura 2: jediný klastr s vysokou dostupností

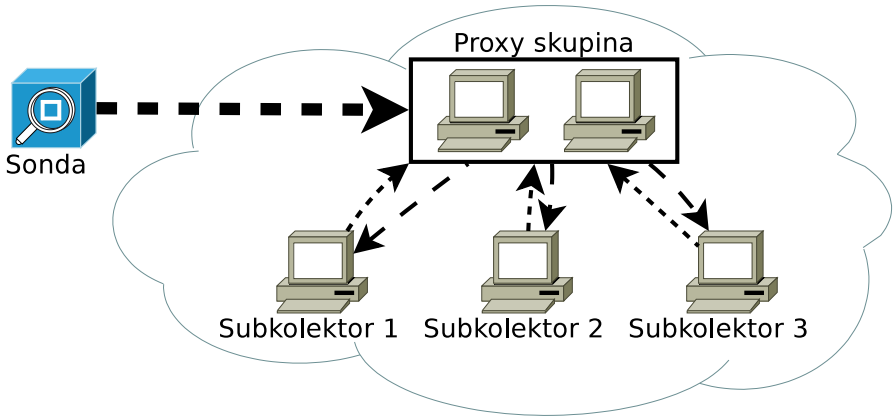
Druhá navrhovaná architektura se snaží odstranit hlavní záporné vlastnost první architektury. Omezuje proto množství uzlů na co nejmenší počet a jak lze vidět níže, vysokou dostupnost lze do současné architektury dostat i bez přidání jediného uzlu.

Architektura 2 (obrázek 5.3) se od té současné liší pouze v jediném místě. Proxy uzel je nahrazen **proxy skupinou**, kterou tvoří minimálně dva proxy uzly. Pro zbytek systému (exportér a subkolektory) je skupina zcela transparentní a jeví se jako jediný proxy uzel. Uzly v rámci skupiny budou v konfiguraci aktivní/pasivní, u které je klíčové rychle detekovat poruchu aktivního uzlu a následně zajistit transparentní rekonfiguraci (tzv. failover). Výpadek subkolektoru bude řešen inteligentním round-robin algoritmem použitým při rozdělování záznamů. Ve chvíli, kdy se aktivnímu proxy uzlu nepovede odeslat záznam na některý subkolektor, algoritmus jej vyřadí a záznamy bude rozdělovat mezi zbylé uzly. Neschopnost podávat korektní výsledky při výpadku subkolektoru (díky historickým datům uloženým lokálně) je potřeba řešit datovou redundancí. Uzly si budou vzájemně uchovávat repliky dat, detailně je proces rozebrán v následující sekci 5.3.

Stejně jako u současné architektury, dva stejné logické prvky nemohou sdílet jeden počítač. Minimální počet fyzických uzlů v klastru ale nevzrostl: dva subkolektory, první zároveň v roli aktivní proxy a druhý v roli pasivní proxy.

I tato architektura **eliminuje oba SPOF** a přináší tak vysokou dostupnost. Díky stejnému počtu prvků jako v současné architektuře nejsou zvýšeny náklady na pořízení a následnou správu. Export záznamů na proxy skupinu není potřeba dělat ve dvou kopiích.

Určité **nevýhody** ale přináší i tato architektura. To, co jsme v arch. 1 vyřešili redundancí hardwaru, je nyní potřeba řešit programově. Failover proxy uzlů, inteligentní round-robin, redundance lokálně uložených záznamů, to vše bude potřeba implementovat. Dalším problémem jsou šablony (u NetFlow v9 a IPFIX), které do proxy uzlu vnášejí stavovost. Od exportérů je ale přijímá pouze aktivní uzel, po rekonfiguraci by nový aktivní uzel šablony neměl a nevěděl by, jak záznamy interpretovat. Mezi aktivním a pasivním uzlem je proto potřeba implementovat sdílení stavu.



Obrázek 5.3: Architektura 2: vysoká dostupnost v jediném klastru.

5.3 Úložiště flow dat

Víceprocesorové systémy lze na základě sdílených komponent rozdělit na tři architektury [57]:

Se sdílenou pamětí: více procesorů sdílí centrální paměť.

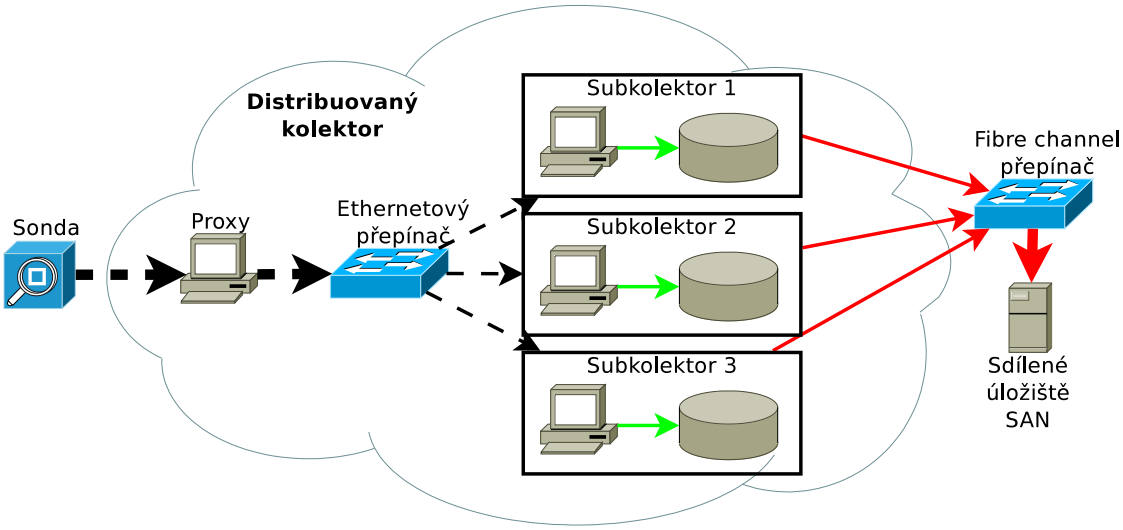
Se sdíleným úložištěm: každý procesor má vlastní paměť, sdílí ale úložiště.

Bez sdílených komponent: ani paměť ani úložiště se mezi procesory nesdílí.

Se sdílenou pamětí se běžně setkáváme u procesorů sdílejících čip (vícejádrové procesory), nebo základní desku (více soketů). Pokud ale procesory nesdílí ani jedno, je situace složitější, protože je potřeba vytvořit jeden logický adresní prostor přes více oddělených fyzických pamětí. Vzniká tak *distribuovaná* nebo *virtuální sdílená paměť* [42]. Pokud ale nejsou procesory propojeny speciálními vysokorychlostními nízkolatenčními propoji (např. NUMAlink od SGI [21]), může se stát sdílená paměť úzkým hrdlem systému. Návrh ale cílí na komoditní hardware, který žádnými takovými propoji nedisponuje, proto jsem sdílenou paměť do návrhu nezahrnoval. Následující odstavce přinesou přehled možností, jak je možné řešit úložiště v klastru počítačů a to jak se sdíleným diskem tak bez něj.

5.3.1 Sdílení množiny disků

V oblasti sdílených úložišť existují **dvě konvenční a široce používané technologie**. Jde o *storage area network (SAN)* a *network-attached storage (NAS)*. **SAN** poskytuje přístup k úložišti na úrovni bloků tak, že se operačnímu systému jeví jako lokálně připojené. Jednotlivá zařízení, která síť SAN tvoří, nemusí být homogenní a může tak jít o kombinaci diskových polí, páskových úložišť, optických úložišť, ... Tato zařízení se připojí do speciálního přepínače, do kterého se připojí také uzly, jenž chtějí úložiště využívat (viz červená cesta na obrázku 5.4). SAN se zaměřuje na vysoký výkon a nízkou latenci, často se z důvodu zvýšení propustnosti používají optické spoje (Fibre Channel). Připojení jednoho blokového zařízení více klienty současně ale znemožňuje použití standardních souborových systémů jako jsou EXT4, XFS nebo NTFS. Souběžné připojení (za použití žurnálování i při připojení pouze ke čtení) by u nich mohlo způsobit různé inkonzistence, protože na takové použití zkrátka nejsou konstruovány. Z toho důvodu vznikly souborové systémy, které se



Obrázek 5.4: Dvě možné realizace úložiště flow dat. Zelené šipky představují tok dat při použití lokálních diskových jednotek (architektura bez sdílených komponent), červeně je zobrazena SAN síť (sdílené úložiště pro všechny uzly). Čárkované čáry představují nezpracované záznamy o tocích (např. IPFIX), plné čáry jsou záznamy zpracované do formátu vhodného k uložení.

specializují na SAN úložiště. Jako příklad uvedu OCFS[2] nebo GFS2[3]. Ty přidávají mechanizmy pro řízení souběžného přístupu, poskytují konzistentní pohled na data a zamezují poškození/ztrátě dat při provádění operací více klientů třeba na stejném souboru.

U technologie **NAS** se nejedná o celou vlastní síť, ale o jedno zařízení, které k úložišti poskytuje přístup na úrovni souborů. Typicky se jedná o specializovaný server obsahující jednu či více diskových jednotek organizovaných do diskových polí. Výhodou je možnost využít standardních souborových systémů, protože klientům není úložiště přístupné na úrovni bloků. Pro přístup k souborům je proto potřeba využít některý k tomu určený protokol, např. NFS, SMB/CIFS nebo AFP.

5.3.2 Bez sdílených komponent

„Počítačový klastr nemůže existovat bez sdíleného úložiště. Je zcela jasné, že klastr je nepoužitelný, pokud jeho členové mají přístup pouze k jejich lokálním úložištím a datům.“ [50] Distribuovaný kolektor je ale případ u kterého autorovo tvrzení nemusí platit, **sdílené úložiště totiž není nezbytně nutné**. Stačí splnit následující podmínku: všechny operace pracující s daty (tzn. vkládání a čtení) se budou vykonávat na všech subkolektorech současně. Proxy uzel obstarává rozdělení dat mezi subkolektory (např. algoritmem round-robin), všechny subkolektory tak současně zapisují. Každý z nich potom na svém lokálním úložišti bude uchovávat $\frac{1}{N}$ záznamů, kde N je celkový počet subkolektorů, všechny dohromady tak nesou kompletní množinu (viz zelená cesta na obrázku 5.4). Speciálně navržený systém řízení dotazování potom může zajistit, že analytická část kolektoru bude pracovat vždy s úplnými daty i bez sdíleného úložiště (všechny subkolektory současně čtou).

[2] https://oss.oracle.com/projects/ocfs/

[3] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Global_File_System_2/index.html

Takto lze jednoduše implementovat architekturu bez sdílených komponent, která je velice dobře škálovatelná. Navíc není potřeba budovat drahou SAN síť, která vyžaduje optickou síťovou kartu na každém uzlu, specializovaný přepínač a samozřejmě alespoň jedno externí úložiště. **Chybí ale odolnost proti výpadku**, protože každý záznam je uložen pouze na lokálním úložišti jednoho ze subkolektorů. Ukládání příchozích záznamů není problém ani při výpadku, proxy uzel jednoduše vynechá nedostupný subkolektor z procesu rozdělování dat. Problém ale nastává při dotazování, jelikož záznamy z lokálního úložiště jednoduše nejsou dostupné.

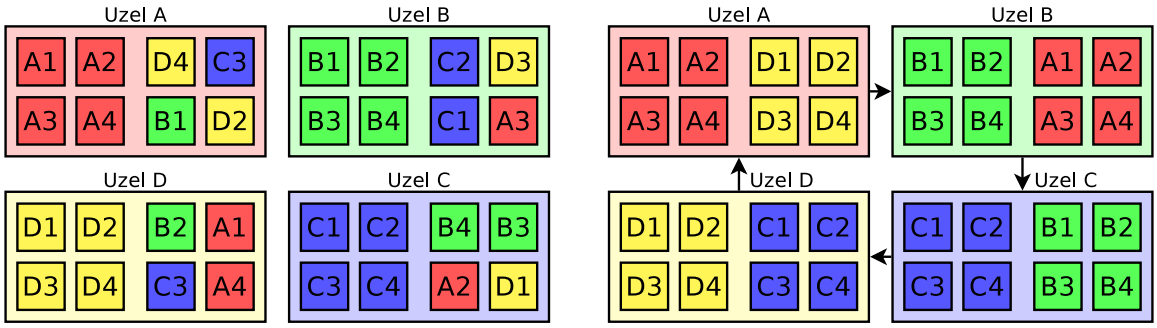
Každý subkolektor je tedy SPOF pouze díky datům, která lokálně ukládá, protože při jeho poruše nastává jejich nedostupnost. Eliminace spočívá v **replikaci** každého záznamu v rámci klastru, lokální úložiště každého subkolektoru tak nebude uchovávat pouze $\frac{1}{N}$ z celkového počtu záznamů, ale minimálně ještě jeden stejný díl jiných záznamů. Dohromady tak každý subkolektor bude uchovávat $\frac{R}{N}$ záznamů, kde R je *replikační faktor*, $R = \{x \in \mathbb{N} | 1 \leq x \leq N\}$. $R = 1$ znamená žádná replikace, naopak $R = N$ znamená, že každý subkolektor bude lokálně uchovávat všechny záznamy, které na klastr z exportérů přišly. Ani jeden extrém ale není reálně příliš použitelný, bez replikace ztrácíme nárok na vysokou dostupnost a v opačném případě dojde k degradaci výkonu a velkému nárůstu objemu uložených dat. Výhodný je replikační faktor dva nebo tři, při kterém bude každý záznam uložen v rámci klastru dvakrát nebo třikrát, přičemž každá kopie na jiném subkolektoru. Přehled možností, jak replikaci implementovat popisují následující odstavce.

Strategie rozmísťování replik

Je možné volit dvě strategie, podle kterých budou repliky záznamů rozmístěny po klastru. První, zvaná **prokládání** (**striping**), rozděluje repliky dat jednoho uzlu rovnoměrně mezi všechny ostatní (obrázek 5.5a). Děje se tak na úrovní souborů nebo bloků, např. známý big data framework *Hadoop*[4] rozděluje soubory na bloky pevné velikosti (typicky 64 MB), které následně rovnoměrně rozmisťuje na jednotlivá lokální úložiště. Výhoda tohoto řešení se projevuje ve chvíli, kdy nastane porucha na jednom uzlu. Repliky souborů/bloků, za jejichž zpracování byl zodpovědný nedostupný uzel, jsou rozprostřeny mezi všechny zbývající uzly, které si tak mohou práci rozdělit rovnoměrně. Nevýhodou je nutnost udržovat databázi fyzického umístění souborů/bloků, Hadoop pro tento účel používá NameNode, který uchovává výhradně metadata.

Druhá strategie vyžaduje logicky uzly uspořádat jako orientovaný cyklický graf, kde každý uzel je vrchol se vstupním i výstupním stupněm rovným jedné. Zjednodušeně řečeno, uzly jsou uspořádány do **kruhu** (obrázek 5.5b). Každý uzel tak má právě jednoho následníka a právě ten uchovává repliku veškerých dat daného uzlu (repliku dat následníka uchovává následník následníka atp.). Tuto strategii používá třeba analytická databáze *Vertica*[5], kde se pro ni používá název *K-safety*. Replikační faktor určuje, kolik následníků uzlu ponese repliku všech jeho lokálních dat. Při $R = 2$ tak replika lokálních dat každého uzlu bude uložena na lokálním úložišti jeho následníka, při $R = 3$ bude navíc uložena replika na následníkovi následníka. Lze tak už při $R = 2$ docílit odolnosti proti poruše až poloviny uzlů v klastru, navíc umístění replik je dané topologií a k jejich nalezení není potřeba databáze. Pevné umístění dat je ale zároveň nevýhodou, protože při poruše uzlu za něj budou muset všechnu práci odvést následníci, kteří mají kopii jeho dat.

[4] http://hadoop.apache.org/
[5] http://www.vertica.com/



(a) Prokládání. (b) Kruhová topologie.

Obrázek 5.5: Strategie rozmísťování replik. Každý uzel nese množinu vlastních souborů a množinu replik cizích souborů.

Naivní replikace kopírováním souborů

Nejjednodušší řešení je průběžně kopírovat databázové soubory na jiný subkolektor (výběr dle zvolené strategie). Tato metoda využívá specifické vlastnosti dat na kolektoru zvané WORM (Write Once Read Many), která říká, že se záznam po vytvoření už nebude modifikovat. Pokud je databázový soubor se záznamy pravidelně rotován (jak je popsáno v sekci 3.3), zapisuje se vždy pouze do jednoho „živého“ souboru. Historické soubory se už budou pouze číst a můžeme je proto jednorázově přenést na lokální úložiště jiného subkolektoru aniž bychom porušili konzistenci. Záznamy z živého souboru je možné přenést až po rotaci (soubor se změní na historický), při poruše lokálního disku proto o záznamy z živého souboru nenávratně přijdeme (jejich množství závisí na délce rotačního intervalu). Zároveň je po návratu uzlu do klastru nutné jej uvést do konzistentního stavu. To znamená umístit na něj repliky historických dat, která na funkčních uzlech vznikla v době jeho poruchy.

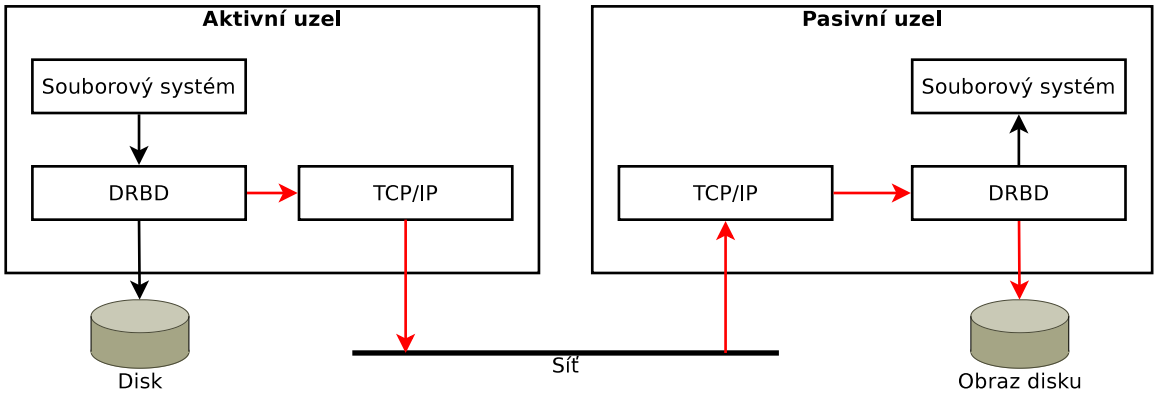
Replikace proxy uzlem

Proxy uzel typicky pošle vstupní záznam na jeden ze subkolektorů. Pokud by ale záznam poslal na dva či více subkolektorů (v závislosti na replikačním faktoru), byla by tak zajištěna redundance. Musí se ale odlišit originální záznamy od replik aby nevznikaly duplicity. Toho lze docílit spuštěním dvou instancí IPFIXcolu na každém ze subkolektorů, jeden pro příjem originálních záznamů a druhý pro příjem replik. To ovšem představuje velkou nevýhodu, protože dvě běžící instance IPFIXcolu by pro uzly představovaly zbytečnou zátěž.

DRBD

Pro operační systém Linux existuje technologie *distribuovaného replikovaného blokového zařízení*[6] (*distributed replicated block device, DRBD*). Ta zajišťuje, že celé blokové zařízení fyzicky umístěno na jednom (aktivním) uzlu, je přes síť zrcadleno na jiné blokové zařízení na druhém (pasivním) uzlu. Na DRBD tak lze nahlížet jako na **síťový RAID 1**. Princip fungování (viz obrázek 5.6) je následující: Do standardního V/V subsystému operačního systému je mezi mezipaměť a diskový plánovač přidána mezivrstva DRBD. Na aktivním uzlu musí každá V/V operace na zrcadleném bloku přes tuto vrstvu projít. Čtecí operace

[6] http://www.drbd.org/



Obrázek 5.6: Zjednodušené schéma vstupně/výstupního subsystému operačního systému s přidanou vrstvou DRBD. Šipky znázorňují tok dat při zápisu na aktivním uzlu, černé představují standardní operace, červené ukazují operace přidané použitím DRBD.

pro ni nejsou zajímavé, každý zápis je ale odchycen a jeho kopie putuje přes síťové rozhraní na pasivní uzel. Pasivní uzel ze síťového rozhraní kopii zápisové operace přepošle na DRBD vrstvu, která se už postará o její provedení na fyzický blok. Jak aktivní tak pasivní uzel provedly stejnou zápisovou operaci a blokové zařízení se zrcadlí. DRBD řeší poruchu jednoho z uzlů, výpadky sítě a disponuje také synchronizačním algoritmem, který po výpadku přenáší pouze ty bloky, které byly skutečně změněny.

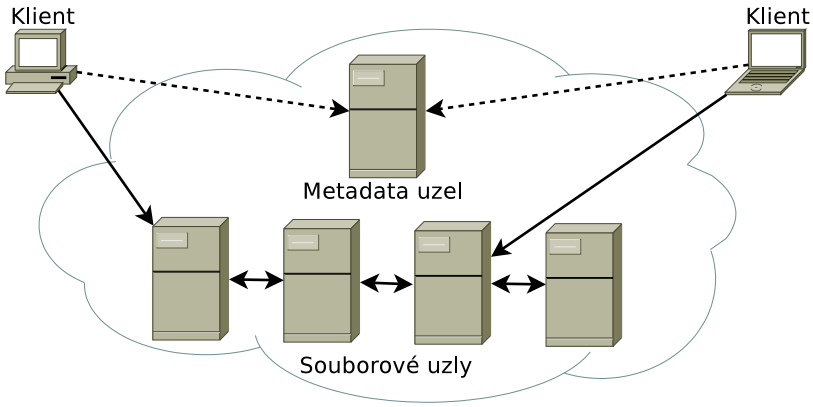
Je potřeba, aby každý uzel měl dvě stejně velká bloková zařízení, jedno v aktivním a druhé v pasivním stavu. Aktivní by sloužilo pro ukládání primárních dat a bylo by zrcadleno na pasivní zařízení svého následníka. Druhou podmínkou je použití kruhové topologie, prokládání DRBD neumožňuje.

Distribuované souborové systémy

Klastrové souborové systémy zmiňované výše umožňují současný přístup několika klientům k jednomu blokovému zařízení. V architektuře distribuovaných souborových systémů (Distributed File Systems, DFS) je blokové zařízení nahrazeno **logickým svazkem** a distribuovanost spočívá v tom, že logický svazek v sobě může zahrnovat lokální úložiště uzlů klastru ale i jakákoliv externí úložiště. Logický svazek se tak rozpíná přes distribuovaný systém a tvoří jednotný jmenný prostor, jednotný bod připojení. Pro klienta se systém snaží být co nejvíce transparentní, snaží se uvést do praxe všechny formy transparentnosti obecného distribuovaného systému uvedené v sekci 4.1 (transparentnost lokace, replikace, selhání, ...). Jako příklad uvedu GlusterFS[7], Ceph[8] nebo Lustre[9]. Terminologie v této oblasti není úplně sjednocená, nejčastěji pro tento typ souborových systému najdeme označení distribuované, ale např. zmiňovaný Lustre se označuje jako paralelní souborový systém. Považuji tyto dva pojmy za ekvivalentní.

Na obrázku 5.7 je zobrazena typická **architektura** DFS, která se skládá ze tří prvků: metadata uzel, datové uzly a klienti. Obláček představuje logický svazek, který v sobě ukrývá jeden či více souborových uzlů a jeden metadata uzel. Prerekvizita každé souborové operace je dotaz na metadata uzel (čárkovaná šipka). Z odpovědi se klient dozví, se kterým

[7] https://www.gluster.org/
[8] http://ceph.com/
[9] http://lustre.org/



Obrázek 5.7: Typická architektura distribuovaného souborového systému.

souborovým uzlem má komunikovat a potom už nic nebrání přímému přenosu dat (plná šipka).

Nepopiratelnou **výhodou** těchto souborových systémů je škálovatelnost. Mohutnost množiny souborových uzlů může sahat od jedničky až po tisíce a protože je zátěž rovnoměrně rozdělována mezi všechny. S každým dalším uzlem navíc roste nejen celková kapacita svazku, ale také výkon. Zároveň DFS počítají s možností výpadku uzlu a nabízí proto automatickou replikaci souborů, transparentní zotavení z výpadku a také uvedení uzlu do konzistentního stavu po návratu do klastru. Replikace je typicky prováděna už při zapisování, při poruše lokálního disku tak nebudou ztraceny ani záznamy z živého souboru. Všechny problémy, které bychom museli řešit při naivní replikaci kopírováním, za nás řeší DFS.

Naopak **nevýhodou** může být centralizace všech metadat do jednoho uzlu. Ten se jednoduše může stát úzkým hrdlem, bez něj je dokonce nedostupný celý svazek (jedná se o SPOF). Některé DFS tento problém řeší duplikací metadata uzlu, existuje ale i elegantnější řešení (viz GlusterFS v sekci 6.3).

Kapitola 6

Implementace a použité technologie

Ze dvou navrhovaných architektur distribuovaného kolektoru jsem zvolil **variantu č. 2: jediný klastr s vysokou dostupností** zajištěnou čistě pomocí softwaru. Hlavním důvodem jsou finanční náklady realizace kolektoru, které nejsou u zvolené architektury navyšovány redundantním hardwarem. Cenou za nižší náklady je ale nutnost redundanci zajišťovat vzájemně mezi uzly v klastru, což klade vyšší nároky na programovou implementaci.

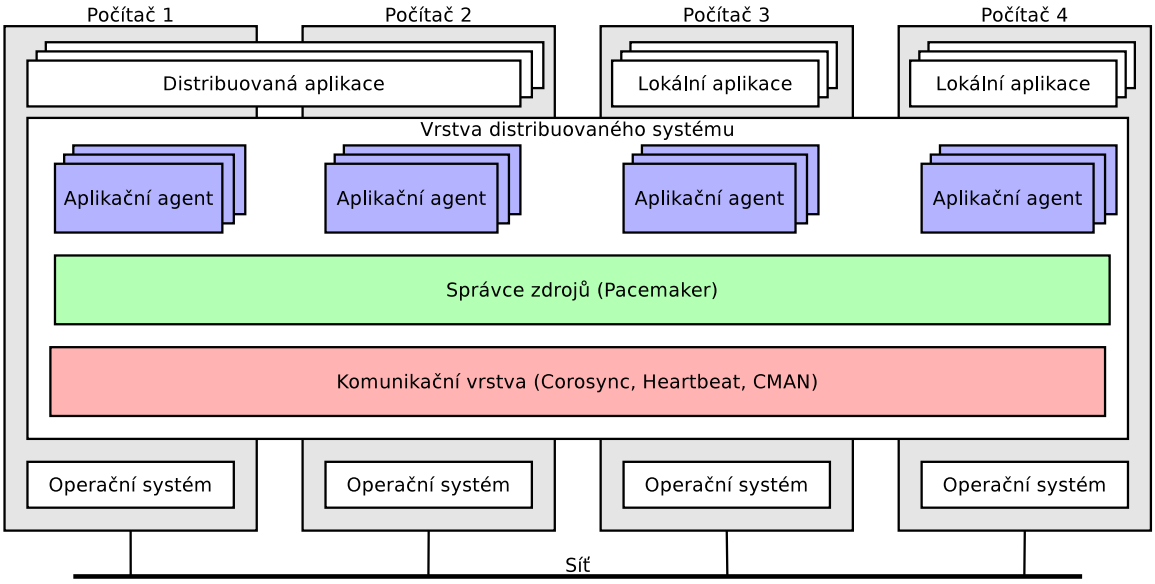
Bylo potřeba řešit celou řadu problémů, při svém hledání jsem nenarazil na žádný volně dostupný softwarový celek, který by je řešil všechny. Zároveň jsem postupem času vyloučil variantu, že bych veškerou potřebnou funkcionalitu implementoval ve vlastní režii, protože to zkrátka nebylo potřeba. Problematikou vysoké dostupnosti v klastru počítačů se zabývá celá řada programů a nástrojů a mým úkolem bylo vybrat jejich kombinaci vhodnou pro tento případ použití.

Výsledný programový zásobník je zobrazen na obrázku 6.1. Mezi programy v jednotlivých vrstvách jsem se snažil vytvářet co nejméně závislostí, aby bylo možné je v případě potřeby jednoduše vyměnit za jiné. Pro komunikační vrstvu v klastru jsem zvolil projekt Corosync (sekce 6.1), jeho služeb využívá správce zdrojů Pacemaker (sekce 6.2). Z velkého množství možností pro uložení a redundanci dat jsem zvolil architekturu bez sdílených komponent, konkrétně distribuovaný souborový systém GlusterFS (sekce 6.3). Na tomto základu staví dvojice programů implementující jádro samotného kolektoru: IPFIXcol a fdistdump, ty jsou popsány v sekcích 6.4 a 6.5.

6.1 Corosync – komunikační vrstva

Komunikační vrstva, někdy nazývána jako klastrová infrastruktura nebo skupinový komunikační systém, je nejnižší vrstvou programového zásobníku použité architektury. Jak již název napovídá, tato vrstva **zajišťuje komunikaci mezi uzly** a dále ji zprostředkovává ve formě aplikačního programového rozhraní vyšším vrstvám. Komunikace uzlů v klastru má svá specifika, která ji odlišují od „běžné“ komunikace v LAN síti. Jde především o potřebu určitých záruk, které TCP/IP protokoly nenabízejí a musíme je proto implementovat na aplikační úrovni. Jako příklad uvedu záruku toho, že pokud je odesilateli zprávy doručena tatáž zpráva zpět, byla doručena také všem ostatním uzlům v klastru.

Dříve si tuto funkcionalitu v případě její potřeby implementovala každá aplikace sama. Tyto implementace byly různě kvalitní, nebyly mezi sebou vzájemně kompatibilní a vzni-



Obrázek 6.1: Programový zásobník použitý při implementaci zvolené architektury.

kaly tak problémy s interoperabilitou aplikací v klastru [23]. Na tento problém zareagovalo konsorcium Service Availability Forum, které v v roce 2003 vytvořilo sadu volně dostupných specifikací aplikačních rozhraní pro dosažení vysoké dostupnosti s využitím klastru počítačů[1]. Následoval vznik projektu OpenAIS[2], který tyto rozhraní implementuje. V lednu 2008 vznikl související projekt Corosync Cluster Engine[3] (dále jen Corosync), který vychází z redukované verze OpenAIS.

Existují ale i jiné programy/knihovny implementující funkcionalitu komunikační vrstvy. Společnost Red Hat pro tyto účely používala program CMAN[4], ve verzi 7 distribuce Red Hat Enterprise Linux jej však nahradil právě Corosync. Podobný osud měl i program Heartbeat[5]. Ten byl nejprve součástí klastrového správce zdrojů Pacemaker (viz sekce 6.2), později ale došlo k rozdělení obou projektů. V současné době umí Pacemaker využívat jak Heartbeat tak Corosync, pro nové instalace se však doporučuje druhý zmiňovaný.

Oblíbenost, splnění veškerých požadavků, stabilita a přítomnost v repozitářích klíčových Linuxových distribucí (Red Hat Enterprise Linux a deriváty, Debian), to jsou důvody které mě vedly k použití Corosyncu jako komunikační vrstvy. Na druhou stranu je potřeba zmínit, že dokumentace k projektu Corosync je ve špatném stavu a při snaze o porozumění principů příliš nepomohou ani chybějící komentáře ve zdrojových kódech. Jediným uceleným zdrojem informací, který se mi podařilo dohledat, je 8 let starý článek [23]. Většina informací v této sekci proto pochází z něj a z manuálových stránek, aktuální stav věcí jsem hledal ve zdrojových kódech, ve verzovacím systému[6] nebo ve zprávách vývojářů v elektronických konferencích.

[1] http://www.saforum.org
[2] https://oss.oracle.com/osswiki/OpenAIS.html
[3] http://corosync.github.io/corosync/
[4] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/High_Availability_Add-On_Overview/ch-cman.html
[5] http://linux-ha.org/wiki/Heartbeat
[6] https://github.com/corosync/corosync

6.1.1 Architektura

Interní architektura Corosyncu je složena z několika vrstev a jak už to u vrstvových modelů bývá, každá jeho komponenta/vrstva je zaměnitelná. Pro tento text jsou ale podstatné pouze dvě komponenty: protokol Totem a uzavřená skupina procesů.

Klíčovou součástí Corosyncu je **implementace protokolu Totem** [4]. Hlavním cílem tohoto protokolu je poskytnout aplikacím jednak spolehlivé, úplně uspořádané doručování zpráv a jednak služby členství. Implementován je také komunikační model rozšířené virtuální synchronie, více je o této problematice uvedeno v sekci 4.4.3. Tyto vlastnosti zajišťují, že uzly v klastru budou synchronizovány i když některý z nich selže nebo když přibude uzel nový. Totem umožňuje použití IPv4 i IPv6, na transportní vrstvě používá protokol UDP. Standardně komunikace probíhá pomoci multicastu, je ale možné využít jak broadcast, tak unicast.

Uzavřená skupina procesů poskytuje rozhraní pro zasílání zpráv mezi procesy, které se do dané skupiny v minulosti přihlásily. Skupina může obsahovat libovolné množství procesů, stejně tak každý proces se může přihlásit do libovolného počtu skupin. Od chvíle přihlášení bude proces přijímat zprávy, které do skupiny poslal nějaký člen skupiny, včetně svých vlastních zpráv. Zároveň může proces zprávy vytvářet. Všichni členové dostávají také notifikace o změně konfigurace skupiny, tedy informace ohledně přihlášení a odhlášení jiných procesů (záměrné i způsobené selháním). Všechny zprávy obsahují ID procesu, který zprávu odeslal nebo způsobil změnu konfigurace. Tato služba využívá protokolu Totem a tedy i komunikačního modelu rozšíření virtuální synchronie, oproti jeho přímému použití ale poskytuje programové rozhraní vyšší úrovně.

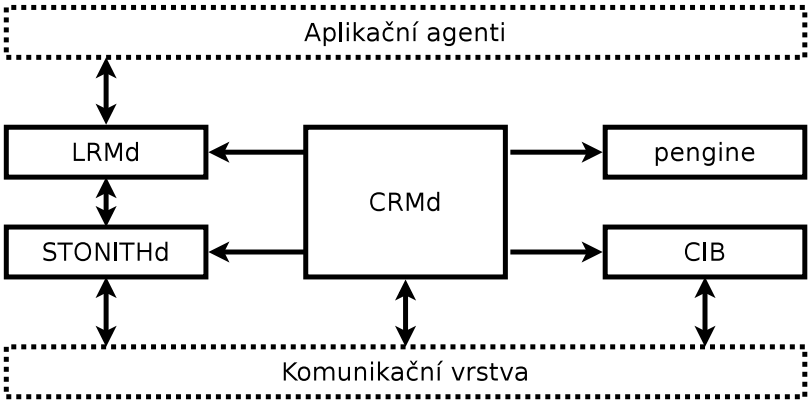
6.1.2 Bezpečnost

Corosync předpokládá, že komunikace mezi uzly bude probíhat pouze v rámci lokální sítě a nebude vystavena do Internetu. I v lokální síti se ale můžou vyskytovat hrozby, proto je možné nastavit zabezpečení všech zpráv protokolu Totem. Pro zajištění důvěrnosti se používá **symetrická kryptografie**, sdílený tajný klíč musí být uložen na každém uzlu. Šifrovaná je celá aplikační vrstva, zprávy odchycené na síti budou bez znalosti tajného klíče nečitelné. O autentizaci a integritu se stará **HMAC** (Hash-based Message Authentication Code), který každou zprávu obohatí o autentizační kód. Corosync je díky tomu odolný vůči útokům zahrnujícím podvržení nebo upravení zprávy.

6.2 Pacemaker – správce zdrojů

Správce zdrojů je druhá vrstva programového zásobníku, přímo využívá služeb komunikační vrstvy a zároveň ovládá aplikační agenty. Úkolem správce zdrojů v zásobníku je **starat se o software**, který v klastru běží. Jak bylo řečeno, vysoká dostupnost u architektury 2 je potřeba řešit programově a právě tuto funkcionalitu přináší do programového zásobníku správce zdrojů.

V prostředí na Unixu založených operačních systémů se používají tzv. *init* systémy. Init program se typicky spouští jako první proces při startu systému (po jádru) a jeho úkolem je spustit všechny ostatní programy [64]. V hierarchickém řazení tak jsou všechny procesy přímými nebo nepřímými potomky init programu. Za běhu systému je s jeho pomocí možné procesy monitorovat, restartovat nebo např. zastavovat, při vypnutí systému je init program zodpovědný za ukončení procesů, které spustil. Mezi další schopnosti takových



Obrázek 6.2: Komponenty klastrového správce zdrojů Pacemaker [20]. Plnou čarou je nakresleno jádro, tečkovanou okolní vrstvy.

programů patří také spouštění procesů v pořadí dle nakonfigurovaných závislostí. To vše je ale limitováno použitím pouze v rámci jednoho počítače, init si není žádným způsobem vědom stavu služeb na jiných uzlech v klastru.

Existují ale také programy, které implementují v klastru počítačů podobnou funkcionalitu jako init programy implementují na jednom uzlu a nazývají se správci zdrojů, nebo přesněji klastroví správci zdrojů (dále jen CRM podle anglického Cluster Resource Manager). CRM si musí být vědom tím, které uzly jsou pod jeho kontrolou a v jakém jsou stavu. To znamená v prvé řadě vzájemnou spolehlivou komunikaci, proto je správce zdrojů bezprostředně závislý na komunikační vrstvě. Potom je CRM schopen provádět koordinaci, spouštění, zastavování, restartování zdrojů, monitorování selhání zdroje nebo celého uzlu, přesun zdrojů mezi uzly, automatický failover a mnoho dalšího. Díky těmto schopnostem je možné v klastru docílit vysoké dostupnosti téměř libovolného softwaru a to i bez ohledu na zdroj selhání (hardware nebo aplikace).

Etalonem v této oblasti je sada programů a nástrojů, jejíž jádro tvoří program Pacemaker. Byla o něm zmínka už dříve ve spojitosti s programem Heartbeat, protože tyto programy byly původně vyvíjeny jako jeden projekt. Později došlo k rozdělení na dva samostatné projekty, v současnosti ale aktivita kolem projektu Heartbeat spíše klesá. Naopak Pacemaker je stále aktivně vyvíjen a široce používán: tvoří základ doplňku pro vysokou dostupnost v komerčních distribucích Linuxu (RHEL[7], SUSE Linux[8]), používá se také např. v systému řízení letového provozu v Německu [20].

Důvody jeho použití v distribuovaném kolektoru jsou podobné jako tomu bylo u Corosyncu: oblíbenost, robustnost, možnosti přesahující požadavky, stabilita a přítomnost v repozitářích klíčových Linuxových distribucí. Informace v této sekci jsou čerpány převážně z online dokumentace projektu [20].

6.2.1 Komponenty

Funkcionalita Pacemakeru je rozdělena do **několika logických komponent**, viz obrázek 6.2. Spouštění probíhá jediným příkazem `pacemakerd`, který pro každou komponentu vytvoří proces jako svého potomka. I nadále ale `pacemakerd` běží jako démon, monitoruje

[7] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/High_Availability_Add-On_Overview/index.html

[8] https://www.suse.com/documentation/sle_ha/book_sleha/data/book_sleha.html

stav svých potomků a v případě selhání provede jeho restart. Procesy všech komponent musí běžet na všech uzlech v klastru.

CIB (z anglického Cluster Information Base) je komponenta, která se stará o uchování konfigurace a aktuálního stavu klastru. Interně pro tyto účely používá značkovací jazyk XML, přičemž statický obsah (definice zdrojů, uživatelská konfigurace) i dynamický obsah (aktuální stav zdrojů) jsou obsaženy v jediném dokumentu. Výsledný dokument je poměrně špatně čitelný a přestože existuje řada způsobů konfigurace Pacemakeru (viz níže), manuální zásah je občas nevyhnutelný. Obsah CIB je automaticky synchronizován napříč všemi uzly. To s sebou přináší výhodu při správě, protože konfigurace je vždy v konzistentním stavu a je možné ji upravovat na libovolném uzlu.

Další komponentou je **pengine** (z anglického Policy Engine), jejíž vstupem je aktuální stav zdrojů a uživatelská konfigurace (vše obsaženo v CIB). Policy engine stojí za logikou Pacemakeru, jeho úkolem je z dostupných informací vypočítat ideální stav klastru. Produktem je *graf přechodu*, jenž obsahuje seznam akcí, které je potřeba vykonat, aby se klastr dostal ze současného stavu do ideálního stavu. Pengine je po většinu času pasivní, stejně jako celý Pacemaker je totiž spouštěn až při události. To v praxi znamená, že dokud nedojde ke změně obsahu CIB, je stav klastru považován za ideální a žádné akce nejsou prováděny (kromě pravidelného monitorování zdrojů).

Seznam akcí potřebných k přechodu do ideálního stavu je vložen na vstup komponentě **CRMd** (Cluster Resource Manager daemon). Jde o centrální prvek, který zprostředkovává komunikaci mezi jinými komponentami, ale také s CRMd na jiných uzlech. Jeho práce se seznamem instrukcí je prostá, po řadě prozkoumá každou akci ze seznamu a zjistí, zda se bude vykonávat na lokálním, nebo na vzdáleném uzlu. V prvním případě pošle akci do lokálního LRMd, ve druhém případě je s použitím komunikační vrstvy akce zaslána do CRMd vzdáleného uzlu.

Důležitým faktem je, že ať je událost spuštěna na jakémkoliv uzlu, seznam instrukcí je vždy zaslán pouze na jeden konkrétní uzel. Jde o tzv. **koordinátora** (Designated Coordinator, DC), kterého si uzly volí mezi sebou a pro jehož volbu platí všechny podmínky i důsledky kvóra, viz sekce 4.4.1. To dělá ovládání klastru centralizovaným. Celou tuto práci ale provází rovnice centralizovaný systém = SPOF, ani tato situace není výjimkou. Pacemaker (za podpory Corosyncu) po výpadku autonomně zvolí nového koordinátora, dokud se tak nestane, tak není možné v celém klastru provádět žádné akce.

V návaznosti na CRMd je ještě potřeba vysvětlit **LRMd** (Local Resource Manager daemon). Jde o komponentu, která od CRMd přijme akci a pokusí se o její vykonání spuštěním konkrétního aplikačního agenta. Vzápětí čeká na výsledek aby jej mohla oznámit koordinátorovi. V závislosti na výsledku se koordinátor rozhodne, zda pokračovat ve vykonávání grafu přechodu, nebo zda vykonávání ukončit a znovu spustit pengine. V případě úspěšného přechodu do nového stavu se Pacemaker vrací k monitorování zdrojů a čeká na událost.

Poslední komponentou je **STONITHd**, což je démon zajišťující tzv. fencing. Více o této problematice je uvedeno v sekci 4.4.2.

6.2.2 Zdroje

Zdrojem či prostředkem (anglicky resource) se rozumí **služba, kterou Pacemaker spravuje** a zajišťuje její vysokou dostupnost. Služba samotná si přitom nemusí být vůbec vědoma toho, že je spuštěna v klastru, nemusí být (a typicky není) implementována s ohledem na vysokou dostupnost. Bez jakéhokoliv zásahu do existující aplikace je tak možné využít výhody distribuovaného systému, což je také nejčastější případ použití klastrového správce

zdrojů. Pacemaker navíc umožňuje spravovat téměř libovolnou službu, stačí pro ni vytvořit aplikačního agenta (viz níže).

Základním typem zdroje je **primitivní zdroj**. Pacemaker se snaží o to, aby se zdroj nacházel v požadovaném stavu, tedy spuštěn nebo vypnut. Pokud má být spuštěn, tak maximálně na jednom uzlu současně. Který uzel to bude záleží na několika faktorech, při výchozí konfiguraci je však zvolen uzel libovolný. Primitivní zdroje je možné shlukovat do **skupiny zdrojů**. U seskupených zdrojů je zajištěno, že budou spuštěny na stejném uzlu a to sekvenčně v daném pořadí (vypínat se budou v opačném pořadí). Typickým případem použití je seskupení IP adresy a aplikačního serveru, který na zmíněné adrese naslouchá. IP adresa musí být přiřazena některému rozhraní na uzlu, kde běží server a musí tomu tak být už před startem serveru. Z primitivního zdroje i ze skupiny lze vytvořit **klon**, který nedělá nic jiného, než spouští zdroje na více uzlech současně. Jde o základní stavební kámen pro služby v konfiguraci typu aktivní/aktivní.

Všem typům zdrojů lze přiřadit libovolné množství pravidel definující různá **omezení a závislosti**. Prvním typem je *lokační* pravidlo, které určuje konkrétní uzly, na kterých se má služba spouštět. Druhým typem je pravidlo *kolokační*, které zajišťuje vzájemnou soudržnost zdrojů na stejném uzlu i bez striktního omezení jejich lokace na konkrétní uzel. *Závislosti* potom určují vzájemné pořadí, v jakém se budou služby spouštět. Bez nich je vykonáváno vše paralelně, to ale nemusí být vždy žádoucí. Pravidla mohou být buď *povinná* nebo *volitelná*. Pokud se nepovede splnit povinné pravidlo, jsou všechny zahrnuté služby vypnuty a uvedeny do chybového stavu. Pacemaker se snaží vyhovět také volitelným pravidlům, pokud se to ale nezdaří, pokusí se spustit službu i jiným způsobem.

Uvedené zdroje a možnosti jejich konfigurace jsou pouze malou ukázkou z dlouhého seznamu schopností Pacemakeru. Zmínil jsem pouze ty, které používám v konfiguraci kolektoru, abych se na ně mohl později odkazovat.

6.2.3 Aplikační agenti

Aby byl Pacemaker **použitelný pro správu jakékoliv aplikace**, pro interakci s nimi využívá aplikačních agenty. Každý primitivní zdroj musí mít svého aplikačního agenta, který implementuje některé z podporovaných rozhraní. Komponentě LRMd potom stačí znát rozhraní pro komunikaci s aplikačním agentem a ten už musí obstarat ovládání konkrétní aplikace (musí tedy být vytvořený na míru aplikaci).

Pacemaker podporuje několik typů a **rozhraní** aplikačních agentů. Čtyři základní jsou:

- Open Cluster Framework (OCF),
- Linux Standard Base (LSB),
- Upstart,
- systemd.

Doporučuje se ale použití OCF[9] kompatibilních agentů. OCF agent není nic jiného, než spustitelný soubor (typicky Bash skript).

Agent je Pacemakerem spouštěn s jediným argumentem, kterým je název požadované **akce**. Aby agent splňoval specifikaci [43], musí implementovat alespoň akce `start` pro spuštění služby, `stop` pro zastavení služby, `monitor` pro zjištění aktuálního stavu služby a `meta-data` pro výpis informací o službě ve formátu XML, existují ale i další (nepovinné)

[9] http://www.opencf.org/

akce. Argumenty programu jsou agentovi předány pomocí proměnných prostředí. Definované jsou také návratové kódy, podle kterých Pacemaker určí, zda se akce zdařila nebo nikoliv.

6.2.4 Konfigurace a správa

V předchozím textu byla řada aspektů zjednodušena nebo vynechána, proto z něj nemusí být patrné, že konfigurace a následná správa Pacemakeru je komplexní záležitost. Přímá práce s CIB ve formátu XML je sice možná, ale velice nepohodlná a náchylná na chyby. Vznikla proto řada nástrojů, které se snaží tyto úkony zjednodušit a do určité míry i automatizovat.

Jednou možností jsou příkazy `crmadmin`, `crm_attribute`, `crm_node`, monitorovací nástroj `crm_mon`, simulační `crm_simulate` a další s předponou `crm_`. Ty jsou součástí Pacemakeru, pro práci s nimi je ale do určité míry pořád potřeba znát strukturu konfiguračního XML. Na druhou stranu nabízejí široké možnosti především v oblasti správy klastru, zjištění aktuálního stavu nebo příčiny problému.

Vyšší úroveň poskytují konfigurační shelly crmsh[10] a PCS[11]. Ty nabízejí podstatně větší pohodlí a jednoduchost, ovšem za cenu odloučení od některých možností. V repozitářích většiny Linuxových distribucí najdeme crmsh, RHEL jej nahrazuje svým vlastním shellem PCS.

Nejvyšší úroveň konfigurace poskytují grafické uživatelské rozhraní. Webový frontend je přímo součást shellu PCS, dále existuje ještě samostatný projekt HAWK (High Availability Web Konsole)[12].

6.3 GlusterFS – úložiště

Ze všech možných variant uložení dat v klastru rozebraných v sekci 5.3 jsem nakonec zvolil distribuovaný souborový systém. Důvody jsou prosté: DFS zajišťují automatickou replikaci, failover i obnovu dat po výpadku, implementovat tyto věci znovu by proto bylo zbytečné. Nevýhodou DFS je ale centralizace metadat, která tak představují SPOF. S tímto problémem se zajímavým způsobem vypořádal GlusterFS, který tento SPOF úplně eliminuje a jeho architektura je skutečně bez sdílených komponent [31].

Důležitým aspektem souborového systému je také **výkon**. Kvůli režii replikace a vlivu komunikace po síti nelze předpokládat, že DFS bude dosahovat celkového výkonu rovnému součtu výkonů všech datových uzlů. Při výběru jsem se řídil několika publikovanými výkonnostními testy, výsledky jednoho z nich [25] jsou uvedeny v tabulce 6.1. Autor testoval sekvenční i náhodné čtení i zápis na klastru o 20 uzlech, ze srovnání vychází jasně nejlépe právě GlusterFS.

6.3.1 Distribuovaná hašovací tabulka

V DFS GlusterFS jsou na rozdíl od ostatních DFS **metadata distribuovaná** spolu s daty, což je možné díky použití distribuované hašovací tabulky, DHT. Ta se principiálně neliší od standardní hašovací tabulky: vytváří asociativní pole, které mapuje *klíč* na odpovídající *hodnotu* [22, 58]. Z klíče se pomocí *hašovací funkce* vypočítá *index*, který poskytuje informaci o umístění hodnoty. Ve standardní HT jsou hodnoty umístěny např. v poli, index

[10] http://crmsh.github.io/
[11] https://github.com/feist/pcs
[12] http://hawk-ui.github.io/

Testovací program	Operace	HDFS	CephFS	GlusterFS
iozone	sekvenční čtení	193	102	688
	náhodné čtení	39	12	284
	sekvenční zápis	239	51	306
	náhodný zápis	-	14	406
dd	sekvenční čtení	220	126	427
	sekvenční zápis	275	64	268

Tabulka 6.1: Srovnání rychlosti zápisu a čtení tří distribuovaných souborových systémů [25]. Výsledky jsou v MB/s.

proto slouží jako ukazatel na položku tohoto pole. V DHT jsou ale hodnoty rozmístěny na více uzlech, index proto musí poskytnou navíc ještě informaci o tom, na kterém uzlu je požadovaná hodnota uložena.

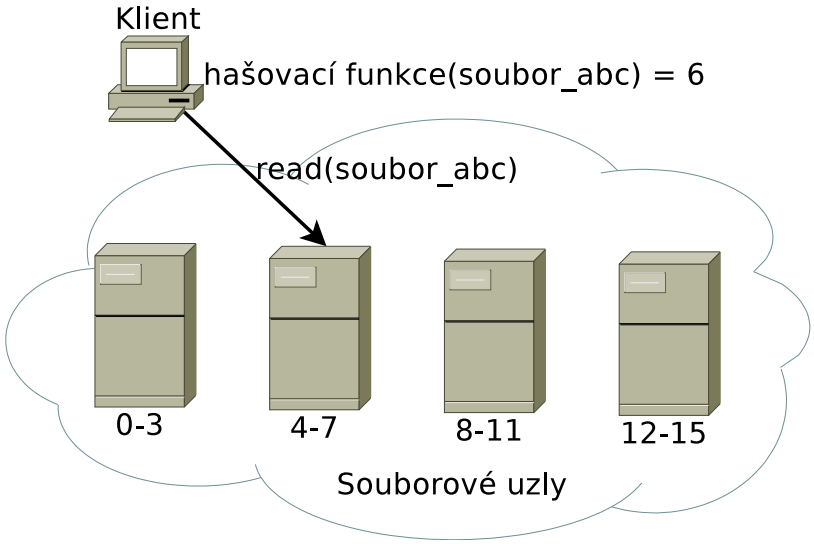
V praxi GlusterFS používá jméno souboru jako klíč, hodnotou je obsah souboru (obrázek 6.3). Rozsah hodnot výsledku hašovací funkce je rozdělen na tolik částí, kolik je v klastru datových uzlů. Každý díl tohoto rozsahu je potom přidělen jednomu uzlu, který bude ukládat všechny soubory, jejichž jméno (konkrétně výsledek hašovací funkce aplikovaný na jméno) spadá do onoho rozsahu. Klientům tak stačí znát rozdělení rozsahu k tomu, aby byli schopni dohledat umístění konkrétního souboru i bez metadata uzlu.

Ne ve všech situacích je ale použití DHT výhodné, především z pohledu výkonu a odezvy. V případě operací jako `open()`, `read()`, `write()` nebo `close()` nad existujícím souborem komunikuje klient jenom s jedním konkrétním uzlem. Pokud zkouší klient přistoupit k souboru, který v úložišti neexistuje, jsou kontaktovány všechny souborové uzly s dotazem, zda náhodou není soubor uložen jinde než kam odkazuje index. To se totiž může stát, když není v době tvorby souboru požadovaný uzel dostupný, např. z důvodu výpadku nebo nedostatku místa na disku a soubor je tak uložen jinam. Dotaz na všechny uzly je nutný také při operacích s adresáři jako `opendir()`, `readdir()` nebo `closedir()`, protože nemají na vstupu jméno souboru. Doba odezvy těchto operací je potom podstatně delší, než při komunikaci s jedním uzlem.

6.3.2 NUFA

Distribuovaná hašovací tabulka je výhodná, když klientský uzel a datový uzel jsou dva rozdílné fyzické systémy (tzn. klient je vzdálený). Je také možné logický svazek DFS připojit na datovém uzlu, potom je klient na stejném fyzickém uzlu jako data (tzn. klient je lokální). Při vytváření nového souboru je lokálním klientem provedena operace hašování jména souboru za účelem hledání, na který uzel soubor uložit, stejně jako v případě vzdáleného klienta. Lokální klient má ale k dispozici lokální úložiště datového uzlu, které je žádoucí využít, protože je to ve většině případů výhodnější (data se nemusí přenášet mezi uzly).

GlusterFS pro takové případy nabízí využití **nerovnoměrné alokace souborů** (Non Uniform File Allocation, NUFA). Při zapnutí NUFA je lokálním úložištím dána větší priorita, při jejich zaplnění nebo nedostupnosti se automaticky přejde zpět k DHT.



Obrázek 6.3: Použití distribuované hašovací tabulku v GlusterFS. Uvedený příklad používá smyšlenou 4 bitovou hašovací funkci.

6.4 IPFIXcol – sběr a uložení záznamů o tocích

Výše zmiňovaný software jako Corosync, Pacemaker nebo GlusterFS je pouze podpůrný: zajišťuje komunikaci v klastru, vysokou dostupnost, failover, nebo úložiště. Konečně se ale dostávám k programům IPFIXcol a fdistdump, které jsou užitečným jádrem kolektoru. IPFIXcol byl v této práci zmíněn už při srovnání způsobů ukládání záznamů a při popisu současného stavu kolektoru, jeho podrobnější popis ale přinášejí až následující odstavce.

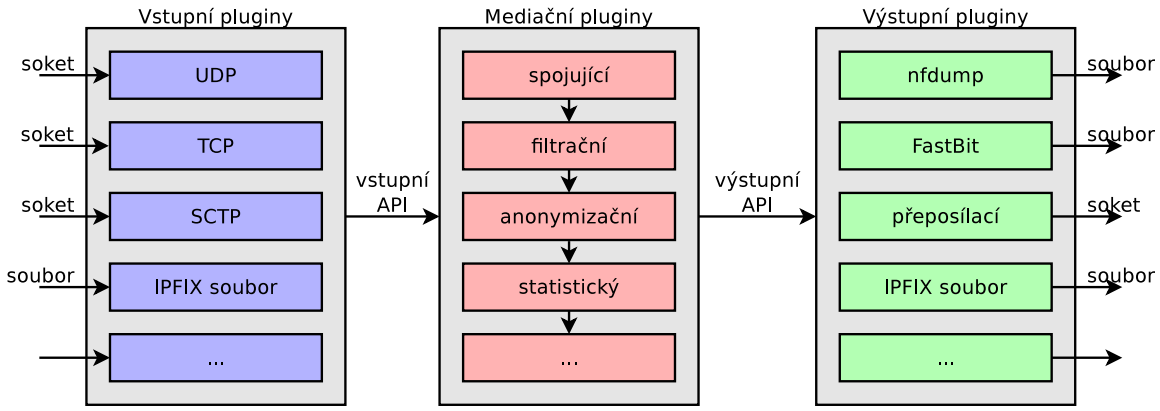
Program IPFIXcol v současné architektuře obstarává celou fázi pro sběr a uložení dat, v nové architektuře nebyl důvod jej nahrazovat. Jde o stabilní software, který sice nebyl vytvořen na míru distribuovanému systému, díky své modulární architektuře však nebyl problém jej patřičně upravit a přizpůsobit. Autorem některých úprav jsem já (UDP vstupní plugin s podporou sdílení šablon, aplikační agent, tvorba PID souboru), jiné potřebné úpravy poskytli správci IPFIXcolu.

6.4.1 Architektura

Tak jako fáze pro sběr a uložení, i IPFIXcol samotný je rozdělen na několik částí [62]:

- jádro,
- vstupní, mediační a výstupní pluginy,
- pomocné nástroje.

Vstupní pluginy přijímají data z rozdílných vstupů a v rozdílných formátech. Lze tak zpracovávat IPFIX i NetFlow formát přicházející přes UDP, TCP, nebo SCTP, existují také vstupní pluginy pro čtení záznamů ze souborů. Každý z těchto pluginů musí data zpracovat a převést do unifikovaného interního formátu, čímž je zajištěna abstrakce vstupu v další fázi zpracování. Takto unifikovaný formát paketu následně prochází všemi specifikovanými **mediačními pluginy**. Pomocí nich je možné data upravovat (např. anonymizovat), filtrovat, spojovat duplicitní toky, generovat různé statistiky atd. **Výstupní plugin** přebírá



Obrázek 6.4: Modulární architektura pluginů IPFIXcolu.

unifikovaný formát záznamů, aby dokončil práci IPFIXcolu. Záznamy je možné ukládat do souborů v různých formátech (IPFIX, nfdump, FastBit, ...), nebo pomocí protokolu IPFIX přeposílat na jiné kolektory. Přeposílací plugin hraje v distribuovaném kolektoru (konkrétně na proxy uzlu) významnou roli, protože se díky němu může IPFIXcol chovat jako exportér a záznamy přicházející na jeho vstupní pluginy dále rozdělovat mezi skupinu subkolektorů. **Jádro** potom řídí všechny pluginy, zajišťuje jim potřebné rozhraní, stará se o uživatelský vstup, signály atp. Tok dat mezi pluginy je znázorněn na obrázku 6.4.

6.4.2 Sdílení stavu mezi instancemi

Nová architektura používá skupinu dvou a více proxy uzlů kvůli rychlému přesměrování práce na záložní uzel při výpadku. Uzly v rámci skupiny jsou v konfiguraci aktivní/pasivní, data (včetně šablon) tedy proudí pouze skrz aktivní proxy instanci. Díky šablonám v NetFlow v9 a IPFIX si ale proces IPFIXcolu musí **uchovávat stav**, aby byl schopný správně interpretovat příchozí záznamy. Stav ale v distribuovaném systému znamená problém, protože bez jeho sdílení mezi proxy skupinou by po výpadku nový aktivní uzel neznal formát příchozích záznamů, a do doby příchodu nových šablon by data zahazoval. To by znamenalo nevratnou ztrátu dat.

Tento problém jsem vyřešil vytvořením nového vstupního pluginu, který vychází z existujícího UDP pluginu a přidává do něj funkcionalitu pro **přeposílání šablon** na pasivní instance IPFIXcolu ve skupině. Využívá k tomu komunikační vrstvy Corosync, konkrétně knihovny implementující uzavřenou skupinu procesů CPG, plugin jsem proto pojmenoval UDP-CPG. Princip fungování je následující: plugin se ve své inicializační funkci přihlásí do skupiny procesů pojmenované „ipfixcol“. Jádro poté v nekonečné smyčce volá přijímací funkci, která v případě UDP čeká na příchod IPFIX/NetFlow paketu ze soketu. UDP-CPG plugin navíc přidává čekání na zprávu z CPG, vše pomocí jednoho vlákna i bez aktivního čekání díky funkci `select()`. Při příchodu paketu z UDP soketu (to je možné pouze na aktivním uzlu) je paket prozkoumán, zda obsahuje nějakou šablonu. Pokud ano, je celý přeposlán multicastem do CPG, dále je zpracováván beze změny spolu s pakety, které neobsahují žádnou šablonu. Při příchodu zprávy z CPG (to je možné na aktivním i pasivním uzlu) jsou z paketu odfiltrovány všechny záznamy jiné než šablonové a takto upravený paket je dále zpracován. Tímto je zajištěno, že aktivní uzel bude zpracovávat všechny pakety, pasivní uzly pouze šablonové, čímž budou všechny instance IPFIXcolu v konzistentním stavu (co se týče šablon). Ve finalizační funkci je provedeno odhlášení ze skupiny „ipfixcol“, tato

funkce se ale vykoná pouze při šetrném ukončení IPFIXcolu. Při pádu aplikace nebo při poruše uzlu zůstane plugin přihlášený. O nápravu se automaticky postará Corosync, který kontroluje živost procesů ve skupině a neživé odhlašuje.

Při výpadku vše probíhá hladce a pro subkolektory transparentně. Proxy skupina se rekonfiguruje a provoz je přesměrován na nový proxy uzel. Ten při prvním paketu z exportéru zahlásí chybu sekvenčního čísla, bez problému ale pokračuje v distribuci záznamů mezi subkolektory a zasílání šablonových paketů do CPG.

6.4.3 Aplikační agent

Jak bylo napsáno v 6.2.3, Pacemaker nepracuje s aplikačním softwarem přímo, ale využívá k tomu aplikační agenty s přesně definovaným rozhraním. IPFIXcol je jeden ze zdrojů, který bude spravován Pacemakerem, aplikační agent pro něj ale neexistoval. Musel jsem jej proto vytvořit.

Pro jeho tvorbu jsem zvolil doporučované **rozhraní OCF**, které povoluje agenty v jakémkoliv jazyce. Většina z existujících agentů ale používá Bash, a proto že se pro tuto úlohu skutečně hodí, zvolil jsem jej i já. Pacemaker potom tento skript spouští s jediným argumentem, kterým je název akce. Pro správu IPFIXcolu stačilo implementovat pouze povinné akce `start`, `stop`, `monitor`, `meta-data` a nepovinnou `validate-all`. Aplikační agent také vyžaduje jeden parametr určující roli, ve které se má IPFIXcol spustit (proxy nebo subkolektor).

Každá role IPFIXcolu může na jednom uzlu běžet pouze v jedné instanci, proto je součástí akce `start` kontrola, zda již požadovaná role běží. Pokud ano, je skript ukončen, v opačném případě je proveden pokus o spuštění. Volání akce `stop` musí danou roli IPFIXcolu ukončit za každou cenu. Nejprve je proveden pokus o řádné ukončení zasláním signálu `SIGTERM`, pokud se proces(y) nepovede ukončit, je zaslán signál `SIGKILL`. Akce `monitor` musí podat zprávu o tom, zda proces(y) dané role běží či nikoliv. Používá k tomu PID soubor, který IPFIXcol při startu vytvoří a při ukončení smaže. Stav procesů z PID souboru je zjištěn a výsledek oznámen Pacemakeru. Akce `validate-all` neprovádí žádnou operaci s procesy, pouze zkontroluje, zda parametry předané skriptu jsou validní. Každý OCF aplikační agent musí také obsahovat svůj popis a definici jednotlivých parametrů ve značkovacím jazyce XML. Tento dokument musí být vypsán na standardní výstup při volání akce `meta-data`.

6.5 Fdistdump – analýza dat

Fdistdump je druhá z aplikací jádra distribuovaného kolektoru a obstarává poslední fázi monitorování síťových toků, kterou je analýza dat. Jde o rychlý, škálovatelný a distribuovaný nástroj, který je na rozdíl od IPFIXcolu od základů **vytvořený pro práci v distribuovaném systému**. Prvním milníkem v jeho vývoji bylo implementovat klíčovou funkcionalitu existujících dotazovacích nástrojů nfdump a fbitdump v DS tak, aby z pohledu uživatele bylo vše transparentní. Velký důraz při vývoji je kladen na krátkou dobu odezvy na jednoduché i komplexní dotazy a flexibilitu dotazování. Program mimo jiné umožňuje záznamy o tocích vypisovat, řadit a agregovat, to vše s možností aplikace syntakticky bohatého filtru. Fdistdump také poskytuje několik úrovní paralelizace: dokáže běžet na několika uzlech současně, na každém uzlu je možné spustit jeden či více procesů a každý proces může používat několik vláken. Cílem této masivní paralelizace je maximální využití výpočetních a vstupně/výstupních zdrojů všech subkolektorů.

Při návrhu distribuovaného kolektoru bylo jasné, že bude potřeba zpracovávat velké množství dat rozprostřených mezi několik uzlů. Bylo proto provedeno několik experimentů s **platformami pro zpracování velkého množství dat** (Hadoop, Vertica, ElasticSearch), včetně nástroje nfdist[13] kombinujícího prvky platformy Hadoop s analytickým systémem nfdump (některé výsledky jsou v [69], zbývající v době psaní práce ještě nebyly publikovány). Žádný z experimentů ale nepřinesl dostatečně uspokojivé výsledky, proto vznikla potřeba vlastní implementace softwaru pro analytickou fázi.

Fdistdump je poměrně **nový projekt**, začal vznikat v létě 2015 z výše popsaných důvodů, staví ale na stabilních základech. Program je kompletně napsán v jazyce C a má dvě softwarové závislosti: první je libovolná implementace standardu komunikačního protokolu Message-Passing Interface (např. Open MPI[14]) a druhou je knihovna pro práci se záznamy o tocích libnf[15]. Jedná se o plně otevřený software, pro vývoj se používá verzovací systém Git a repozitář je volně přístupný na serveru GitHub[16]. Součástí repozitáře je také dokumentace v podobě manuálových stránek a README souboru. Většinovým autorem zdrojového kódu i programové dokumentace jsem já.

6.5.1 Softwarové závislosti

Kvůli požadované schopnosti efektivně pracovat v DS bylo klíčové zvolit správný komunikační protokol. V oblasti superpočítačů se pro tyto účely často používá **Message-Passing Interface** (MPI), které se ukázalo jako vhodné také pro fdistdump. Jde o standardizované rozhraní pro zasílání zpráv a paralelní programovací model, ve kterém se přenášejí data z jmenného prostoru jednoho procesu do jmenného prostoru jiného procesu [28]. Skupina komunikujících procesů není limitována na jeden fyzický počítač, protože MPI podporuje obecně jakýkoliv způsob vzájemné komunikace se zachováním stejné syntaxe i sémantiky operací (reálně se používá sdílená paměť, TCP/IP a Infiniband). Rozhraní je poměrně bohaté, v současné době nabízí komunikaci z bodu do bodu, skupinovou komunikaci, topologie procesů, datové typy nebo třeba paralelní vstup a výstup.

Druhou softwarovou závislostí je **knihovna libnf**, která usnadňuje práci s datovými soubory, záznamy a informačními elementy. Knihovna je v současné době kompatibilní s formátem souborů nfdump, který má ale řadu nevýhod (řádková orientace, nepodporuje položky proměnné délky), uvažuje se proto o použití jiného, více variabilního formátu. Rozhraní knihovny je rozděleno na čtyři skupiny:

- práce se soubory,
- čtení a modifikace záznamů,
- agregace a řazení záznamů,
- filtrace záznamů.

Souborovou část rozhraní tvoří funkce `lnf_open()`/`lnf_close()` pro otevření/zavření souboru, záznamy se následně čtou/zapisují pomocí `lnf_read()`/`lnf_write()`. **Čtení a modifikaci záznamů** umožňuje skupina funkcí `lnf_rec`, nejdůležitější jsou párové funkce `lnf_rec_fget()` a `lnf_rec_fset()` pro čtení a nastavení jednotlivých informačních elementů. Libnf umožňuje využít také hašovací tabulku (funkce `lnf_mem`), která je

[13] https://github.com/vytautas/nfdist
[14] https://www.open-mpi.org/
[15] http://libnf.net/
[16] https://github.com/CESNET/fdistdump

určena pro **agregaci a řazení**. Z uživatelského hlediska stačí nastavit množinu klíčových elementů, množinu neklíčových elementů a jejich agregačních metod (minimum, maximum, suma atp.) a volitelně také směr řazení pomocí funkce `lnf_mem_fadd()`. Při každém vložení záznamu do tabulky funkcí `lnf_mem_write()` knihovna zajistí tvorbu agregačního klíče, aplikaci hašovací funkce a vyhledání klíče v tabulce. Pokud je klíč nalezen, existující a nový záznam jsou sloučeny dohromady za použití nastavených agregačních metod, v opačném případě je nový záznam umístěn do tabulky beze změny. Záznamy lze z tabulky opět získat funkcí `lnf_mem_read()`. Poslední skupina funkcí `lnf_filter()` umožňuje uživateli nastavit **filtr záznamů**, kterým dle hodnot informačních elementů záznam buď projde nebo neprojde.

6.5.2 Architektura a průběh programu

Komunikační model je typu master/slave, který dovoluje programu běžet na téměř libovolném počtu uzlů, stejně dobře ale funguje také na jediném uzlu. Minimální počet procesů jsou dva, jeden řídící (master) a jeden pracovní (slave), s rostoucím počtem procesů přibývají pouze pracovní procesy, řídící zůstává vždy pouze jeden. Díky abstrakci meziprocesové komunikace, kterou poskytuje MPI, nezáleží na tom, zda řídící a pracovní proces běží na stejném uzlu nebo na dvou uzlech vzdálených tisíce kilometrů. V praxi je ale vhodné každý proces umisťovat na dedikovaný uzel v klastru, jenom tak lze dosáhnou maximální efektivity. **Datový model** vychází z modelu MapReduce: zpracování velkých souborů se provádí v místě jejich uložení (fáze Map), na řídící proces se přenáší pouze kompaktní předzpracovaná data (fáze Reduce). Všimněme si, že komunikační i datový model přesně zapadají do architektury distribuovaného kolektoru. Řídící proces je vhodné spustit na uzlu bez dat (proxy) a tam, kde data naopak jsou (subkolektory), je potřeba spustit pracovní proces.

Vysokoúrovňové **blokové schéma operací** programu fdistdump je na obrázku 6.5. Šedý blok znázorňuje proces v rámci kterého je několik operací (barevné bloky). Šipky s celou čárou propojují operace tak, jak jdou za sebou při vykonávání programu, a přerušované šipky propojují bloky komunikující mezi sebou zasíláním zpráv.

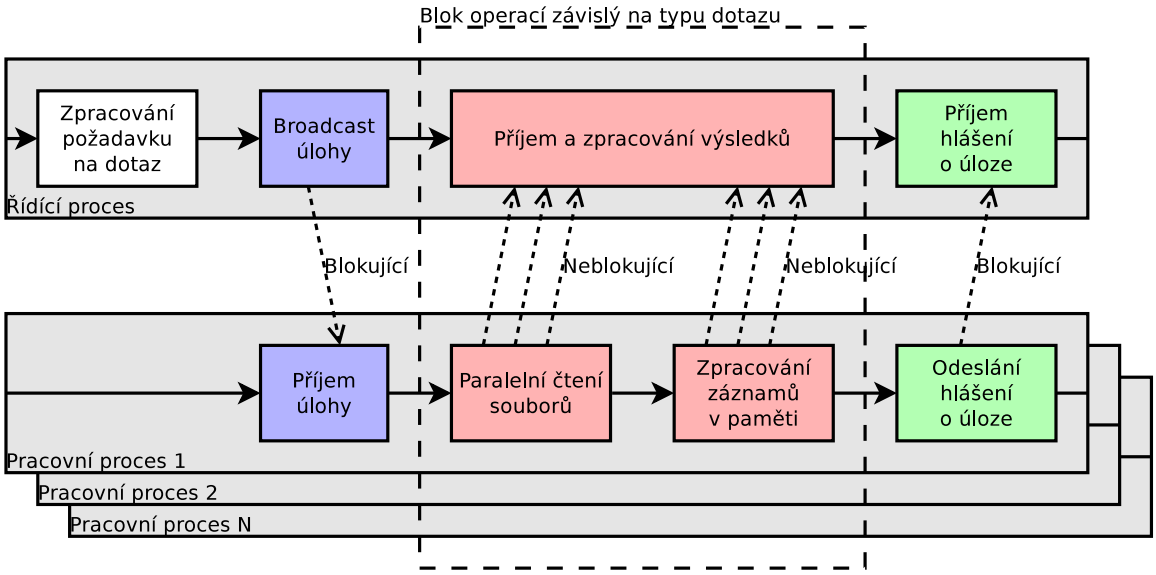
Pracovní procesy po spuštění čekají na zadání úlohy od řídícího procesu, který potřebné informace od uživatele získá z argumentů na příkazové řádce. Řídící proces rozešle úlohu všem procesům pomocí skupinové komunikace metodou `MPI_Bcast()` a potom se uchýlí k čekání na výsledky. Pracovní procesy po příjmu úlohy začnou číst všechny žádoucí soubory a se získanými záznamy budou nakládat dle typu dotazu (viz níže). Nakonec ale vždy pošlou výsledek (třeba i prázdný) zpět na řídící proces, který provede sloučení výsledků všech pracovních procesů. Po příjmu hlášení o úloze od všech pracovních procesů může řídící proces prezentovat výsledky uživateli.

Existují tři **typy dotazů**:

Vypisovací: základní typ dotazu bez jakéhokoliv zpracování záznamů. Každý pracovní proces čte paralelně několik souborů, záznamy neukládá do paměti ale okamžitě odesílá neblokující komunikací na řídící proces, který je vypisuje.

Řadící: každý pracovní proces čte paralelně několik souborů a každé vlákno do svého adresního prostoru ukládá záznamy do jednosměrně vázaného lineárního seznamu. Po přečtení všech požadovaných souborů se provede seřazení seznamů a jejich následné sloučení do jednoho v lineárním čase[17]. Následuje postupné odesílání záznamů z čela seznamu na řídící proces, který provádí slučování seřazených seznamů od všech

[17] V tuto chvíli je slučování implementováno jiným, méně optimálním způsobem.



Obrázek 6.5: Vysokoúrovňové blokové schéma operací programu fdistdump. Celé šipky zobrazují návaznost operací, přerušované šipky představují meziprocesovou komunikaci, tedy zasílání zpráv.

pracovních procesů. Výsledný seznam už není potřeba ukládat, je možné jej rovnou vypisovat a šetřit tak čas i paměť.

Agregační: může nebo nemusí zahrnovat řazení, většinou ale řazení zahrnuje, protože touto kombinací vznikají tzv. *Top-N statistiky*. Průběh agregačního dotazu je podobný tomu řadícímu, jen se místo lineárního seznamu využívá hašovací tabulka. Řídící proces si ale už nemůže dovolit záznamy rovnou vypisovat, je potřeba nejdříve provést kompletní sloučení všech hašovacích tabulek vzniklých na pracovních procesech.

Veškerá **meziprocesová komunikace** (přerušované šipky v blokovém schématu 6.5) je implementována zasíláním zpráv s využitím knihovny MPI. Fdistdump ale z rozsáhlého rozhraní MPI využívá pouze malou část, konkrétně jde o rozhraní pro komunikaci a datové typy. Pro přenos většiny zpráv jsou použity neblokující funkce, díky kterým je možné souběžně provádět výpočet (respektive čtení dat) a odesílání dat na řídící proces. Často používaná je také blokující skupinová komunikace, která je nejen efektivnější než komunikace z bodu do bodu, ale lze ji použít také jako synchronizační nástroj napříč celou skupinou procesů. Při startu programu, kdy musí pracovní procesy čekat na zadání úlohy, je k tomu použita funkce `MPI_Bcast()` a před ukončením, kdy musí pro změnu čekat řídící proces na všechny výsledky, se skupinově volá `MPI_Gather()`.

6.5.3 Rozšíření pro vysokou dostupnost

Standard MPI v aktuální verzi 3.1 neposkytuje žádné mechanismy, které by řešily **odolnost proti poruchám** při běhu programu. Pokud na některém uzlu nastane porucha, chování knihovních funkcí je nedefinované. V praxi jsem se při používání Open MPI setkal se dvěma druhy chování: runtime MPI se buď ukončil s chybovou hláškou, nebo program čekal ve funkci skupinové komunikace. Oba dva druhy chování jsou samozřejmě nežádoucí, protože výpadek musí být pro uživatele transparentní.

V budoucnu rozhraní MPI o tuto funkcionalitu pravděpodobně doplněno bude, v současnosti je ve stádiu návrhu[18]. Některé komerční implementace (od IBM a Intelu) už odolnost proti poruchám implementovaly, ve volně dostupných se tak zatím nestalo. A protože se v návrhu s komerčním softwarem nepočítá, musel jsem pro fdistdump zajistit řešení poruch při běhu ve vlastní režii.

Vytvořil jsem proto **obalovací skript**, který sleduje stav uzlů v klastru a na jeho základě řídí dotazování. Pro sledování jsem využil nástroj `crm_mon`, který je součást Pacemakeru a rychle reaguje na změnu stavu uzlů. Při požadavku na dotaz se tak nejdříve zkontroluje stav klastru a na základě dostupnosti uzlů se vytvoří konfigurace pro MPI a fdistdump. Dotaz se spustí a při jeho běhu se v monitorování stavu průběžně pokračuje. Pokud dojde k výpadku uzlu který je součást dotazu, po uplynutí určitého časového intervalu se dotaz restartuje. Před restartem se vytvoří nová konfigurace pro MPI a fdistdump, díky datové redundanci je možné i přes výpadek až poloviny uzlů stále zajistit korektní výsledek.

Pro lepší představu uvedu **příklad** s dotazem v klastru z obrázku 5.3 (pět uzlů, dva dedikované proxy a tři subkolektory). Uživatel spustí dotaz. Skript zkontroluje stav klastru a zjistí, že všechny uzly běží. Vytvoří tedy konfiguraci pro fdistdump, kde řídící proces umístí na jeden z dedikovaných proxy uzlů a na všech subkolektorech spustí pracovní proces. V průběhu dotazu ale nastane porucha na subkolektoru 2. Obalovací skript čeká několik sekund, pro případ že by šlo pouze o chvilkovou nedostupnost, po uplynutí intervalu ale fdistdump ukončuje. Uživatel je o této skutečnosti informován. Skrip zná strategii rozmísťování replik v klastru a proto ví, že repliku dat subkolektoru 2 uchovává jeho následovník, v tomto případě subkolektor 3. Na základě toho vytvoří novou konfiguraci pro fdistdump, kde řídící proces opět umístí na jeden z dedikovaných proxy uzlů, pracovní proces však spustí pouze na dostupných subkolektorech. Subkolektoru 3 je také předána informace o tom, že musí zastoupit svého předchůdce, tedy zpracovat navíc repliky jeho dat. Potom už skript opět přechází k průběžnému monitorování stavu klastru.

[18] http://fault-tolerance.org/

Kapitola 7

Konfigurace klastru a služeb

Tato kapitola popisuje kroky, které jsem musel provést, abych služby popsané v předchozí kapitole uvedl do provozu a vytvořil tak jeden funkční celek – distribuovaný kolektor. U nastavení služeb a programů uvádím vždy konečný stav, ke kterému jsem se však musel dopracovat řadou průchodů cyklu konfigurace, experiment, vyhodnocení. Konfiguraci jsem se vždy snažil vytvářet obecně, přesto kapitola nemá sloužit jako návod, který by bylo možné následovat pro zprovoznění kolektoru na libovolném klastru. Především v útržcích konfiguračních souborů lze vidět hodnoty specifické pro mnou používanou hardwarovou sestavu, také jsem z prostorových důvodu do útržků nevkládal některé méně důležité parametry.

V rámci vývoje, konfigurace i experimentů jsem primárně používal počítačový klastr o čtyřech uzlech, každý v následující **hardwarové konfiguraci**: jednodeskový počítač ODROID-XU4[1] osazen procesorem Exynos 5422 s architekturou ARMv7, 2 GB operační paměti a gigabitovou Ethernetovou síťovou kartou. Přes rozhraní USB 3.0 byl ke každému uzlu připojen pevný disk Western Digital o kapacitě 1 TB s 5400 otáčkami za sekundu[2]. Síťové propojení obstarával 8 portový gigabitový přepínač TP-LINK TL-SG108[3]. Jako **operační systém** jsem zvolil GNU/Linux v distribuci Ubuntu 15.04 (Vivid Vervet) s jádrem ve verzi 3.10.92-71, která je jako jediná podporovaná ze strany výrobce použitých počítačů.

Před konfigurací služeb programového zásobníku je potřeba nastavit **síť a další podpůrné služby**. Některé jsou vyžadovány, zbývající jsou uvedeny kvůli usnadnění správy klastru. Všechny uzly musí mít přidělenou alespoň jednu statickou IPv4 nebo IPv6 adresu. Síťová komunikace mezi uzly nesmí být blokovaná, důležité je adekvátní nastavení firewallu, pokud se používá. Je dobré uzlům přiřadit také doménová jména a odpovídající záznamy vložit na lokální DNS server nebo do souboru `/etc/hosts`. Nejsem si vědom toho, že by některá z používaných služeb přímo vyžadovala synchronizaci hodin, rozhodně se ale doporučuje je v rámci klastru v synchronizaci mít. Co už ale vyžadováno je, jsou SSH klíče bez hesla. Open MPI používá při spouštění úloh protokol SSH a vyžaduje přístup ze všech uzlů na všechny uzly. Vhodným řešením je vygenerovat si dvojici klíčů pomocí nástroje `ssh-keygen` a následně soukromý i veřejný zkopírovat na všechny uzly, což zajistí také pohodlné ovládání celého klastru. Vyplatí se používat také paralelní SSH, které umožňuje posílat příkazy nebo soubory paralelně na specifikovanou skupinu uzlů.

[1] http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825
[2] http://www.wdc.com/en/products/products.aspx?id=470
[3] http://www.tp-link.com/no/products/details/cat-4763_TL-SG108.html

7.1 Corosync

Corosync musí být nainstalován a neustále spuštěn jako démon na všech uzlech klastru. Běžně se nachází v repozitářích Linuxových distribucí, pro **instalaci** spolu se závislostmi tedy stačí využít správce balíčků. Potom už je možné Corosync spustit příkazem `corosync`. Aby bylo možné spouštět démona pomocí init systému, v některých distribucích je potřeba změnit v souboru `/etc/default/corosync` řádek obsahující `START=NO` na `START=YES`.

Zprávy protokolu Totem lze volitelně **zabezpečit** symetrickou šifrou a autentizačním kódem HMAC. K tomu je potřeba vygenerovat sdílený klíč příkazem `corosync-keygen` a jeho kopii umístit na všechny uzly.

Spolu s instalací dojde také k vytvoření výchozího konfiguračního souboru v `/etc/corosync/corosync.conf`, ta je ale příliš obecná a je potřeba ji na všech uzlech nahradit. **Konfigurační soubor** Corosyncu je rozdělen na bloky, každý blok je pojmenován a je ohraničen složenými závorkami. Jména bloků nejvyšší úrovně jsou `totem`, `logging`, `quorum`, `nodelist` a `qb`, v nich se mohou nacházet další, zanořené bloky. Každý konfigurační příkaz musí být součástí nějakého bloku a musí vyhovovat předpisu `klíč: hodnota`.

Blok `totem` obsahuje konfigurační příkazy pro **protokol Totem**. Ten vyžaduje nastavení verze a pro zapnutí zabezpečení je nutné zvolit hašovací funkci a šifru. Totem umí komunikovat redundantně pomocí více síťových rozhraní, každé z nich vyžaduje konfigurační blok `interface` s exkluzivní hodnotou `ringnumber`, zároveň je potřeba nastavit přesnou adresu na rozhraní nebo číslo sítě, multicastovou adresu a číslo portu. Existuje celá řada dalších konfiguračních možností, jejich výchozí hodnoty jsou ale vyhovující.

```
totem {
  version: 2
  crypto_hash: sha256
  crypto_cipher: aes256

  interface {
    ringnumber: 0
    bindnetaddr: 192.168.2.0
    mcastaddr: 239.255.1.1
    mcastport: 5405
  }
}
```

Pomocí příkazů v bloku `logging` lze nastavit několik cílů **logování** (standardní chybový výstup, soubor a syslog), včetně jejich přidružených možností. V bloku `quorum` je potřeba nastavit algoritmus **kvóra** (dostupný je však pouze jeden) a očekávaný počet hlasů, který by se měl rovnat počtu uzlů v klastru.

```
logging {
  to_stderr: no
  to_logfile: yes
  to_syslog: yes
  logfile: /var/log/corosync/corosync.log
}

quorum {
```

```
  provider: corosync_votequorum
  expected_votes: 4
}
```

Zbývající bloky není potřeba v konfiguračním souboru uvádět, protože jejich výchozí hodnoty jsou vyhovující. Uvedená konfigurace je sice minimalistická, ale využívá nejefektivnější způsob komunikace a zároveň poskytuje důvěrnost, integritu i autentizaci zpráv. Redundanci protokolu Totem jsem nemohl využít, protože jsem měl k dispozici pouze jediné síťové rozhraní na každém počítači. Kontrolu úspěšného spuštění Corosyncu lze provést prozkoumáním logů a příkaz `netstat -gavn` měl potvrdit členství v nastavené multicastové skupině (v mém případě 239.255.1.1).

7.2 GlusterFS

GlusterFS je rozdělen na klientskou a serverovou část, kolektor však využívá obě dvě. Pro **instalaci** obou částí je opět nejjednodušší využít správce balíčků (pro CentOS je potřebný repozitář EPEL) a potom buď spustit démona `glusterd` manuálně, nebo využít init systém. Úspěšné spuštění GlusterFS démona lze ověřit v lozích, nebo příkazem `netstat -tavn | grep "2400[7|8]"` (servery spolu komunikují na portech 24007 a 24008).

Před vytvořením logických svazků je potřeba pro ně připravit prostor na **fyzických discích**, tedy to, co se v terminologii Glusteru nazývá *brick* (česky *cihla*). Jako podkladový souborový systém je možné použít jakýkoliv s podporou rozšířených atributů, doporučuje se však použití XFS. Na zvoleném blokovém zařízení jsem proto vytvořil souborový systém XFS s označením „flow“, vytvořil jsem přípojný bod a zařízení připojil. Na podkladovém souborovém systému je následně nutné vytvořit adresářovou strukturu dle jmenné konvence z dokumentace:

```
mkfs.xfs /dev/sda -L flow
mkdir /data
mount -L flow /data

mkdir -p /data/glusterfs/conf/brick/
mkdir -p /data/glusterfs/flow/brick{1,2}/
```

Všechny uzly, na kterých bude běžet server, musí být součástí tzv. **důvěryhodné množiny uzlů**. V počátečním stavu jsou všechny uzly pouze ve své vlastní množině, příkazem `gluster peer probe UZEL` dojde k sjednocení lokální množiny a množiny uzlu `UZEL`. Postupně tak lze sjednotit všechny uzly do jedné důvěryhodné množiny.

Nyní už je vše připravené pro vytvoření **logických svazků**. Použil jsem svazky dva, jeden pro sdílení konfigurace a druhý pro ukládání záznamů o tocích. Konfigurační svazek „conf“ se rozpíná přes všechny uzly klastru a na každém z nich je jediná cihla v adresáři `/data/glusterfs/conf/brick`. Z důvodu rychlejší reakce na výpadek uzlu je potřeba nastavit hodnotu časového limitu `network.ping-timeout` na hodnotu nižší než výchozích 42 sekund:

```
gluster volume create conf replica 4 UZEL{1..4}:/data/glusterfs/conf/brick
gluster volume set conf network.ping-timeout 10
gluster volume start conf
```

Datový svazek „flow“ se rozpíná pouze přes uzly subkolektoru. Replikační faktor jsem nastavil na 2, jsou proto potřeba také dvě cihly na každém uzlu. Používám replikační strategii, ve které jsou uzly logicky uspořádány do kruhu a následník uchovává repliku veškerých dat svého předchůdce (viz sekce 5.3.2). Primární cihla uzlu 2 je proto replikovaná na sekundární cihlu uzlu 3, primární cihla uzlu 3 na sekundární cihlu uzlu 4 a tak dále až do uzavření kruhu (začíná se uzlem 2, protože první je dedikovaná proxy). Před startem je ještě nutné zapnout NUFA a stejně jako na svazku „conf“, snížit hodnotu časového limitu:

```
gluster volume create flow replica 2 UZEL2:/data/glusterfs/flow/brick1/ \
                                     UZEL3:/data/glusterfs/flow/brick2/ \
                                     UZEL3:/data/glusterfs/flow/brick1/ \
                                     UZEL4:/data/glusterfs/flow/brick2/ \
                                     ...
gluster volume set flow cluster.nufa enable
gluster volume set flow network.ping-timeout 10
gluster volume start flow
```

Nyní je úložiště připravené a stačí oba svazky jen **připojit** („conf“ na všech uzlech, „flow“ pouze na subkolektorech). Pro tento účel musí být vytvořeny adresáře `/data/conf` a `/data/flow`:

```
mkdir /data/{conf,flow}
mount -t glusterfs localhost:/conf /data/conf/
mount -t glusterfs localhost:/flow /data/flow/
```

7.3 Aplikační software

Programový zásobník distribuovaného kolektoru obsahuje dva programy, které tvoří jeho jádro: IPFIXcol a fdistdump. Následující odstavce popisují jejich instalaci, konfiguraci a také metody, kterými jsem ověřoval korektnost obou kroků.

7.3.1 IPFIXcol

Instalace IPFIXcolu je složitější, protože pro něj nejsou k dispozici binární balíčky. Je potřeba nainstalovat všechny závislosti, stáhnout zdrojové kódy a program přeložit:

```
git clone https://github.com/CESNET/ipfixcol.git
cd ipfixcol/base
autoreconf -i
./configure --prefix=/usr/
make && make install
```

Po instalaci jsem na sdíleném logickém svazku „conf“ vytvořil adresář `ipfixcol` s funkcí sdíleného úložiště všech **konfiguračních souborů** IPFIXcolu. Pro instance v roli proxy jsem vytvořil konfiguraci `startup-proxy.xml`, která obsahuje definici jednoho vstupního pluginu UDP-CPG (pro sdílení stavu) a jednoho přeposílacího výstupního pluginu (pro rovnoměrnou distribuci záznamů mezi subkolektory). Pro subkolektory je konfigurace umístěna v `startup-subcollecotor.xml` a definuje jeden vstupní plugin UDP (obyčejný příjem od proxy) a jeden výstupní plugin lnfstore (uložení ve formátu nfdump pomocí knihovny libnf).

Dále je do sdíleného adresáře `/data/conf/ipfixcol/` možné umístit konfigurační soubory `internalcfg.xml` a `ipfix-elements.xml`.

Správnost konfigurace je vhodné **ověřit** spuštěním IPFIXcolu v obou rolích a kontrolou správnosti výstupů. Následně je možné programy ukončit, protože se později o jejich běh bude starat správce zdrojů Pacemaker.

```
ipfixcol -c /data/conf/ipfixcol/startup-proxy.xml \
         -i /data/conf/ipfixcol/internalcfg.xml -v 2

ipfixcol -c /data/conf/ipfixcol/startup-subcollector.xml \
         -i /data/conf/ipfixcol/internalcfg.xml -v 2
```

7.3.2 fdistdump

Ani pro fdistdump nejsou k dispozici binární balíčky, jeho dvě závislosti MPI a libnf je proto nutné předem nainstalovat. Implementace Open MPI a/nebo MPICH v repozitářích bývají, stačí tak využít správce balíčků. Libnf se musí přeložit ze zdrojových kódů:

```
git clone https://github.com/VUTBR/nf-tools.git
cd nf-tools/libnf/c/
./prepare-nfdump.sh
autoreconf -i
./configure --prefix=/usr/
make && make install
```

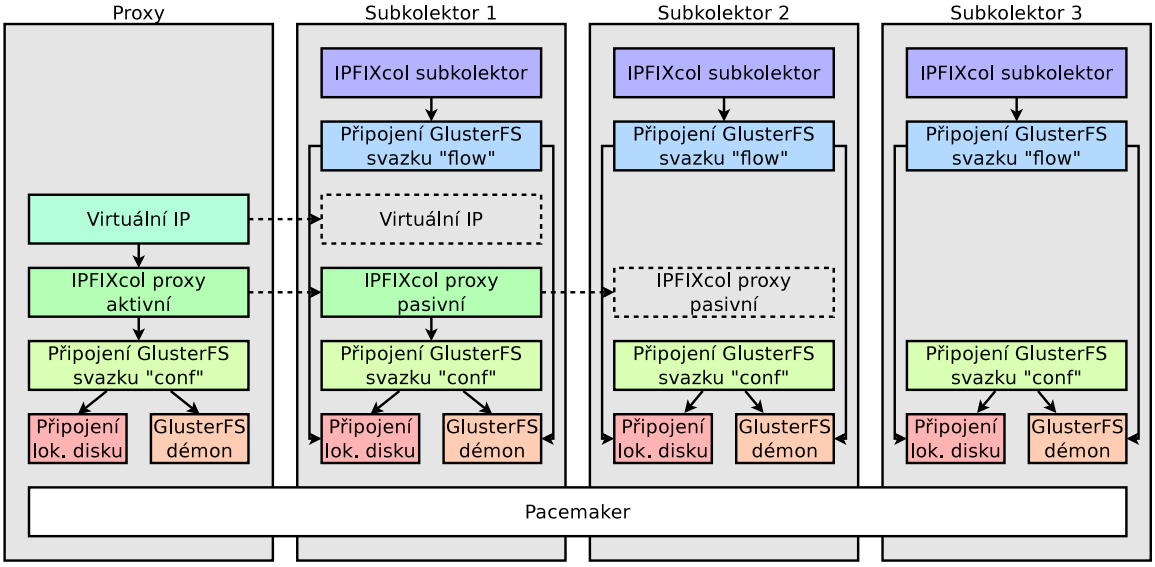
Potom už nic nebrání tomu přeložit a **nainstalovat** fdistdump samotný:

```
git clone --branch develop https://github.com/CESNET/fdistdump.git
cd fdistdump
autoreconf -i
./configure --prefix=/usr/
make && make install
```

Konfigurace MPI typicky obnáší tvorbu tzv. *hostfile* souboru, který obsahuje seznam všech uzlů, na kterých má program být spuštěn. Při spouštění dotazů přímo přes fdistdump je hostfile určitě vhodný, jelikož se seznam uzlů nemusí psát při každém spuštění (stačí zadat cestu k hostfile souboru). Při spouštění dotazů přes obalovací skript, což je preferovaná varianta, se ale tento seznam uzlů tvoří dynamicky na základě jejich aktuální dostupnosti, hostfile se proto nepoužívá. V konečném důsledku tak není potřeba MPI, libnf ani fdistdump nijak konfigurovat.

7.4 Pacemaker

Když je připraven všechen další software, je možné přejít k poslednímu bodu realizace programového zásobníku, kterým je instalace a konfigurace Pacemakeru. Jak bylo napsáno v sekci 6.2.4, pro Pacemaker existuje řada nástrojů, které jeho konfiguraci usnadňují. Jedním z nich je i shell crmsh, který jsem pro konfiguraci používal já. Také Pacemaker i crmsh se běžně nacházejí v repozitářích Linuxových distribucí, pro **instalaci** tedy stačí využít správce balíčků. Ke spuštění všech potřebných procesů na pozadí je možné přímo spustit příkaz `pacemakerd`, lepší je ale využít init systém. Úspěšné spuštění všech procesů lze ověřit



Obrázek 7.1: Konkrétní programový zásobník na klastru o čtyřech uzlech. Plné šipky znázorňují závislosti, tečkované jsou přesuny zdrojů při výpadku.

v lozích nebo příkazem `corosync-cpgtool`, který vypisuje členství v uzavřené skupině procesů.

Konfigurace v mém případě začínala vypnutím jakékoliv snahy o **fencing** v podobě STONITH, protože mé počítače nepodporovaly IPMI, ani jsem neměl k dispozici dostatečně chytré PDU. Do konfiguračního XML Pacemakeru je možné přidávat jednotlivým uzlům jakékoliv **textové atributy**. Toho jsem využil k rozlišení subkolektorů od dedikovaných proxy a také k přiřazení následníků subkolektorům. Každý subkolektor má atribut s klíčem „successor“ a hodnotou v podobě identifikátoru uzlu následníka. Dedikované proxy je možné identifikovat tak, že tento atribut nemají nastavený:

```
crm configure property stonith-enabled=false

crm node attribute UZEL2 set successor UZEL3
crm node attribute UZEL3 set successor UZEL4
crm node attribute UZEL4 set successor UZEL2
```

Dalším krokem je **definice zdrojů**, konkrétně primitiv a jejich klonů. Cílový stav se snaží znázorňovat obrázek 7.1, který vychází z obecného programového zásobníku (obr. 6.1) a dále jej upřesňuje. Definice primitivního zdroje začíná klíčovým slovem `primitive`, následuje jednoznačný identifikátor (pro lepší přehlednost jim dávám předpony `prim_`) a použitý aplikační agent ve tvaru `rozhraní:třída:název`. Volitelně lze zdrojům nastavovat parametry (klíč. slovo `param`) meta parametry (`meta`) a operace (`op`). Parametry jsou různé pro každého aplikačního agenta, operace jsou globální a každé lze nastavit maximální dobu běhu (`timeout`), operaci `monitor` se musí navíc nastavit i monitorovací interval.

Celkový počet **primitivních zdrojů** v kolektoru je sedm a lze je rozdělit do tří kategorií. První kategorii (odstíny červené na obrázku 7.1) tvoří připojení lokálního disku `prim_local_mount` a démon GlusterFS `prim_glusterd`. Druhou kategorii (odstíny zelené) tvoří připojení logického svazku „conf“ `prim_gluser_conf_mount`, proxy instance IPFIXcolu `prim_ipfixcol_proxy` a virtuální IP adresa `prim_virtual_ip`. Třetí kategorii (od-

stíny modré) tvoří připojení logického svazku „flow“ `prim_gluser_flow_mount` a subkolektorové instance IPFIXcolu `prim_ipfixcol_subcollector`. Operace a jejich parametry jsou u všech zdrojů podobné, z prostorových důvodů je uvádím pouze u prvního zdroje.

```
#kategorie 1:
primitive prim_local_mount ocf:heartbeat:Filesystem \
        params device="-L flow" directory="/data" \
        params fstype=xfs force_clones=true \
        op start timeout=60 interval=0 \
        op stop timeout=60 interval=0 \
        op monitor timeout=40 interval=20
primitive prim_glusterd ocf:glusterfs:glusterd

#kategorie 2:
primitive prim_gluster_conf_mount ocf:heartbeat:Filesystem \
        params device="localhost:/conf" directory="/data/conf" \
        params fstype=glusterfs
primitive prim_ipfixcol_proxy ocf:cesnet:ipfixcol.sh \
        params role=proxy \
        params startup_conf="/data/conf/ipfixcol/startup-proxy.xml" \
        params internal_conf="/data/conf/ipfixcol/internalcfg.xml" \
        meta migration-threshold=1 failure-timeout=350
primitive prim_virtual_ip ocf:heartbeat:IPaddr2 \
        params ip=192.168.2.21

#kategorie 3:
primitive prim_gluster_flow_mount ocf:heartbeat:Filesystem \
        params device="localhost:/flow" directory="/data/flow" \
        params fstype=glusterfs
primitive prim_ipfixcol_subcollector ocf:cesnet:ipfixcol.sh \
        params role=subcollector \
        params startup_conf="/data/conf/ipfixcol/startup-subcoll.xml" \
        params internal_conf="/data/conf/ipfixcol/internalcfg.xml"
```

Z primitivních zdrojů je následně potřebné vytvořit **klony**, které zdroje spustí na více uzlech současně. Oba dva zdroje první kategorie spolu s připojením logického svazku „conf“ musí být spuštěny paralelně na všech uzlech (výchozí chování klonu). IPFIXcol v roli proxy je také možné spustit na všech uzlech, ale bohatě stačí ho spustit pouze na dvou (parametr `clone-max=2`), aby mohla být jedna instance aktivní a druhá pasivní. Zdroje třetí kategorie je třeba spustit na všech subkolektorech, vytvořil jsem proto klon přes všechny uzly a explicitně zakázal jeho přítomnost na dedikovaném proxy uzlu pomocí lokačního pravidla (viz níže).

```
clone clon_local_mount prim_local_mount
clone clon_glusterd prim_glusterd

clone clon_gluster_conf_mount prim_gluster_conf_mount
clone clon_ipfixcol_proxy prim_ipfixcol_proxy \
        params clone-max=2
```

```
clone clon_gluster_flow_mount prim_gluster_flow_mount
clone clon_ipfixcol_subcollector prim_ipfixcol_subcollector
```

Zbývá už jen nastavit pravidla určující **chování, omezení a závislosti zdrojů**. Definice každého pravidla obsahuje skóre, což je celé číslo s rozsahem $(-\infty, \infty)$ a určuje důležitost pravidla. Zjednodušeně řečeno, všechny hodnoty jiné než $-\infty$ a ∞ jsou pouze doporučení, které se Pacemaker snaží splnit, v případě neúspěchu ale pokračuje dále. Skóre s hodnotou nekonečno dělá pravidlo povinným, bez jeho splnění nemůže Pacemaker pokračovat a zdroj podléhající pravidlu nespustí nikde.

Lokační pravidla (klíč. slovo `location`) Pacemakeru říkají, na který uzel má daný zdroj umístit. Jedno povinné lokační pravidlo jsem aplikoval na IPFIXcol v roli subkolektor, který se tak spustí pouze na uzlech, které mají definován atribut `successor`. Druhé pravidlo, tentokrát volitelné, jsem aplikoval na IPFIXcol v roli proxy, který tak preferuje běh na dedikovaném proxy uzlu, v případě jeho nedostupnosti se ale spustí i na subkolektoru. **Kolokační pravidla** (klíč. slovo `colocation`) zajišťují vzájemnou soudržnost zdrojů na stejném uzlu. V konfiguračním útržku uvádím pouze dvě, ve skutečnosti jich ale pro správnou funkci klastru musí být podstatně více. Např. uvedené povinné kolokační pravidlo pro virtuální IP adresu určuje, že virt. IP musí být přiřazena pouze uzlu, na kterém je spuštěna instance IPFIXcolu v roli proxy. Poslední jsou **řadící pravidla** (klíč. slovo `order`), které definují vzájemné pořadí spouštění služeb. V útržku uvádím pouze dvě, první zajišťuje připojení lokálního disku před připojením logického svazku „conf“, díky druhému je vždy nejprve připojen logický svazek „flow“ a až potom je spuštěn IPFIXcol v roli subkolektor.

```
location loca_gluster_flow_require_designated clon_gluster_flow_mount \
        rule -inf: not_defined successor
location loca_ipfixcol_proxy_prefer_designated clon_ipfixcol_proxy \
        rule 100: not_defined successor

colocation colo_gluster_conf_with_glusterd inf: \
        clon_gluster_conf_mount clon_glusterd
colocation colo_virtual_ip_with_ipfixcol_proxy inf: \
        prim_virtual_ip clon_ipfixcol_proxy

order orde_local_mount_before_gluster_conf \
        clon_local_mount clon_gluster_conf_mount
order orde_gluster_flow_before_ipfixcol_subcollector \
        clon_gluster_flow_mount clon_ipfixcol_subcollector
```

Kapitola 8

Experimenty a vyhodnocení

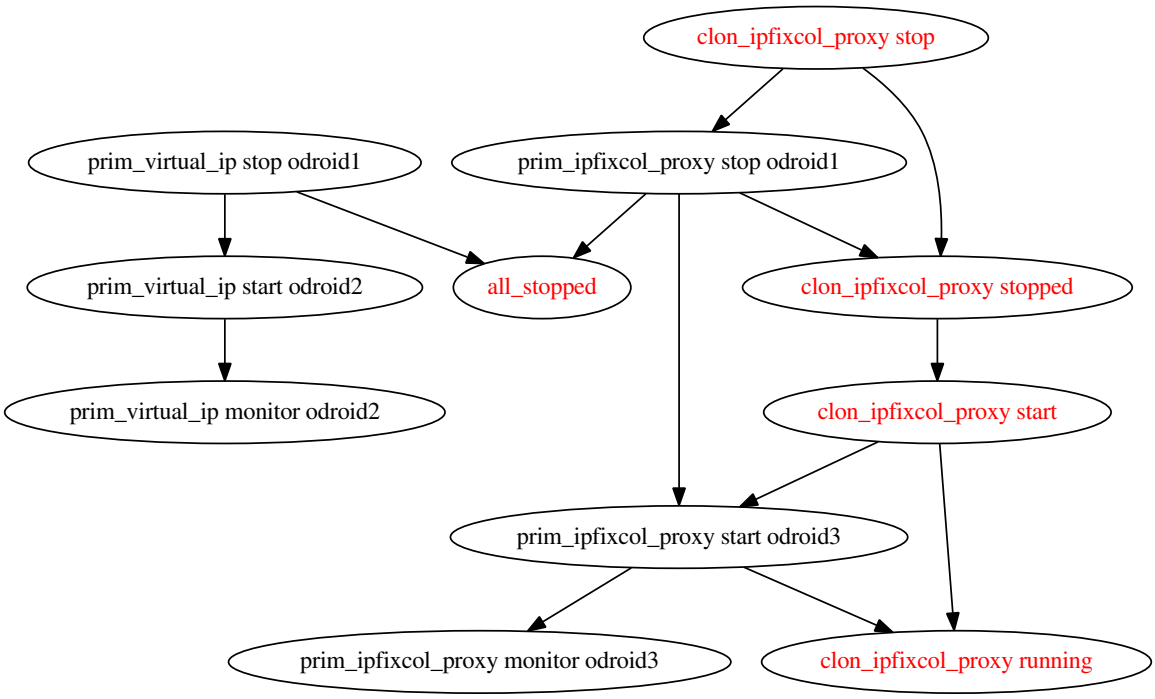
Tato kapitola ověřuje výsledky veškeré práce od návrhu nové architektury, přes výběr a implementaci softwaru až po konfiguraci klastru a použitých služeb. V sekci 8.1 je popsáno experimentální ověřování vysoké dostupnosti a spolehlivosti systému jako celku, v následující sekci 8.2 jsou testy ukládací částí kolektoru. Na analytickou část a řízení dotazování se zaměřuje sekce 8.3. V závěru kapitoly se také nachází celkové shrnutí a vyhodnocení výsledků.

8.1 Poruchy a výpadky

Prvním krokem obnovy systému po selhání nějaké komponenty je zjištění poruchy, a právě **doba trvání detekce poruchy** tvoří významnou část celkové doby uvedení systému do operativního stavu. Na aplikační úrovni je situace jednoduchá, protože každý primitivní zdroj Pacemakeru má přiřazenou monitorovací operaci, kterou tento správce zdrojů pravidelně spouští v předem daném časovém intervalu. Čas detekce selhání aplikace se tak bude pohybovat od nuly až po zvolený monitorovací interval. O detekci výpadků na úrovni uzlů se stará Corosync, u kterého jsem rychlost změřil experimentálně. Časy jsem zjistit z logovacích souborů, přesnost je proto pouze v jednotkách sekund, pro tento experiment to ale plně dostačuje. Časový limit má Corosync ve výchozím nastavení poměrně krátký, protože už sekundu po výpadku hlásil změnu konfigurace. Související zpráva byla zaslána do všech CPG skupin a v Pacemakeru to vyvolalo výpočet nového stavu. Čtyři sekundy po výpadku byl dostupný seznam akcí nutných pro přechod do nového stavu a další tři sekundy trvalo jejich vykonání. Volba koordinátora je rychlá, protože jeho výpadek (a tedy i nutnost volby nového koordinátora) se na výsledném času nepodepsal.

Druhým krokem obnovy po selhání je **vykonání akcí**, které systém uvedou do operativního stavu. Akce a jejich pořadí závisí na definici zdrojů, přidružených pravidel a nastavení Pacemakeru. Chování při různých druzích poruch jsem ověřoval ve dvou krocích: nejprve simulací pomocí příkazu `crm_simulate` a poté zavedením skutečné poruchy, výsledky by měly být shodné. Výstupem simulace je orientovaný graf akcí potřebných k přechodu klastru do nového stavu. Vrcholy představují akce a hrany znázorňují jejich závislosti: akce bez vstupních hran nemají žádné závislosti a je možné je okamžitě vykonat, akce se vstupními hranami musí počkat na dokončení všech svých závislostí. Graf je možné vizuálně reprezentovat (viz obrázek 8.1), kde jsou černým textem napsány skutečné akce (budou zaslány do LRMd) a červeným textem jsou napsány pseudoakce pro zjednodušení grafu.

Jako příklad uvedu experiment, který měl za cíl ověřit chování Pacemakeru při **selhání**



Obrázek 8.1: Graf akcí potřebných pro přechod klastru do nového stavu při selhání aktivní proxy instance IPFIXcolu.

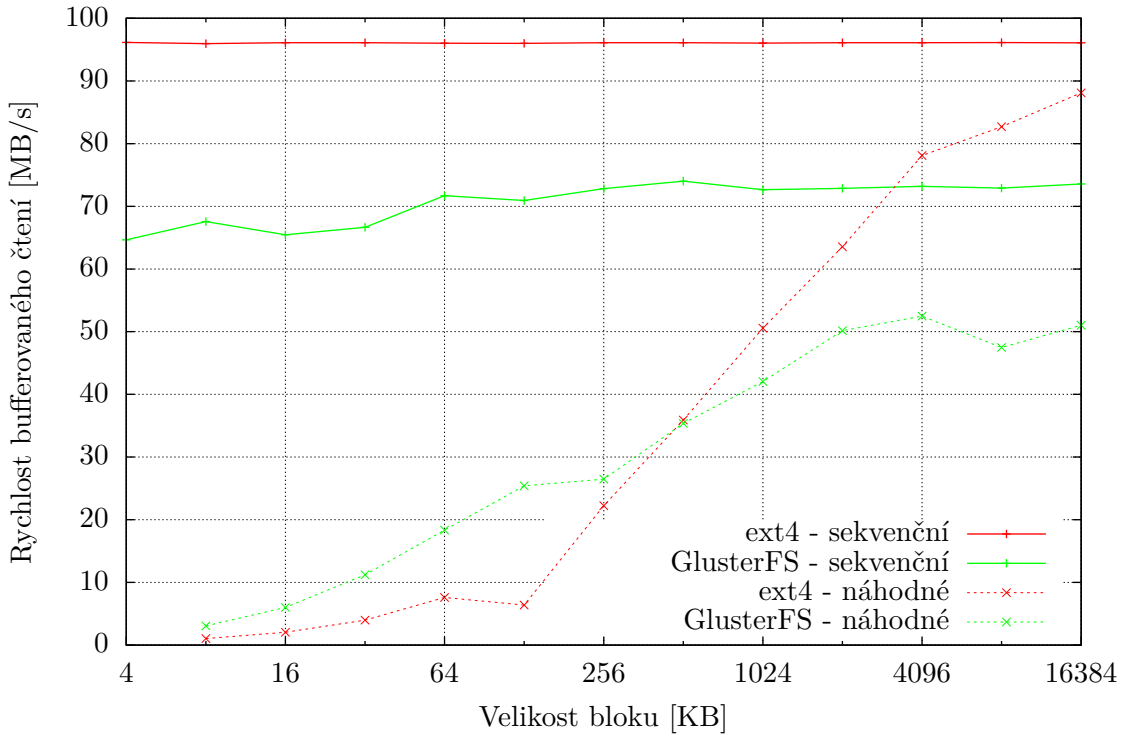
aktivní proxy instance IPFIXcolu. Simulaci jsem provedl příkazem `crm_simulate -L -S -i "prim_ipfixcol_proxy_monitor_20000@odroid1=7"`, který do řídící logiky vloží monitorovací operaci s chybovým návratovým kódem. Výstup příkazu je na obrázku 8.1 a lze na něm pozorovat dva hlavní úkony. Prvním úkonem je přesun virtuální IP adresy na uzel `odroid2`, kde doposud běžela pasivní proxy instance IPFIXcolu (tímto se z ní stane aktivní instance). Druhým úkonem je spuštění nové pasivní instance na uzlu `odroid3`. Graf odpovídá očekávání, nastavení je tedy správné.

8.2 Výkonnost úložiště

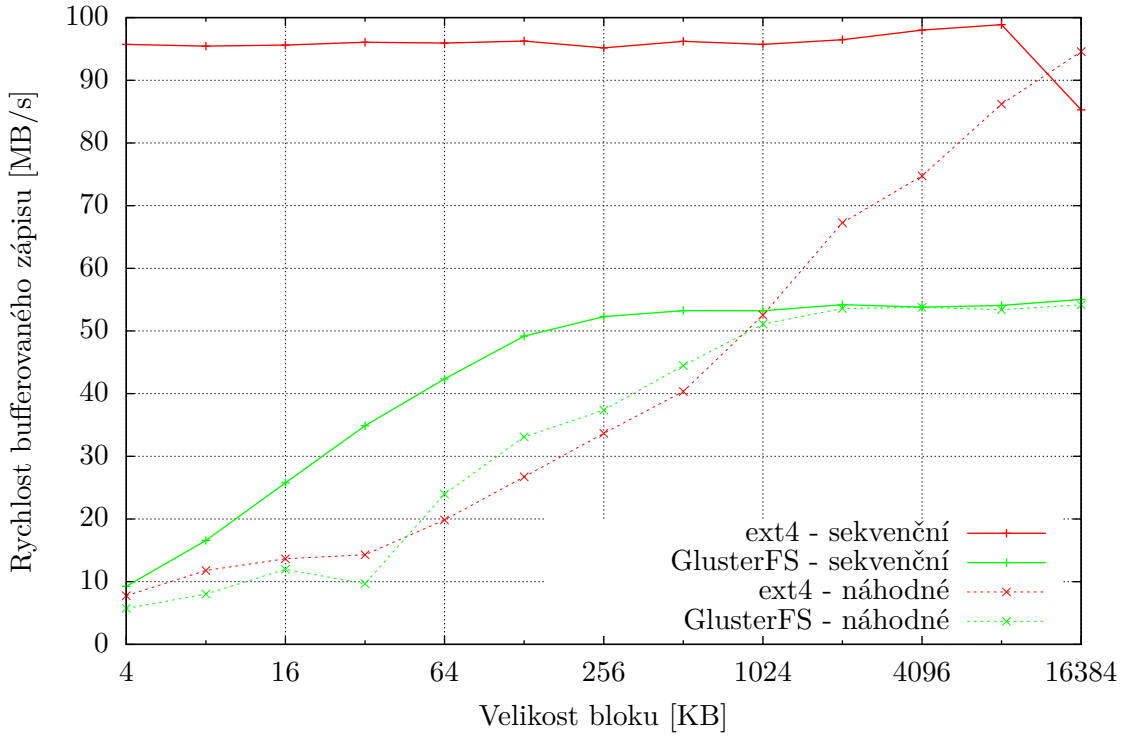
Existuje řada článků, ve kterých jejich autoři měří výkonnost souborového systému GlusterFS z pohledu vzdáleného uživatele, který jej využívá jako cloudové úložiště dat. Nová architektura kolektoru ale nasazuje GlusterFS specifickým způsobem, protože každý server je zároveň klient a vstupně/výstupní operace díky technologii NUFA směřují převážně na lokální úložiště. Kvůli automatické replikaci souborů na uzel následníka se ale všechny zápisové operace současně musí přenést po síti a vykonat i vzdáleně. Zajímala mě režie, která tímto vzniká, proto jsem provedl výkonnostní srovnání GlusterFS se standardním souborovým systémem ext4. GlusterFS má ale také schopnost automatické synchronizace stavu po obnovení uzlu z výpadku, proto jsem experimentálně ověřil jakou dobu trvá, než lze po výpadku uzel zase plnohodnotně používat.

8.2.1 Čtení a zápis souborů

Tento experiment srovnává souborové systémy ext4 a GlusterFS z hlediska rychlosti čtení a zápisu souborů. Ext4 je běžný diskový souborový systém, který pracuje s jediným lokálním



Obrázek 8.2: Rychlost synchronního čtení s využitím vyrovnávací paměti.



Obrázek 8.3: Rychlost synchronního zápisu s využitím vyrovnávací paměti.

diskem a neposkytuje žádnou redundanci dat. Oproti tomu GlusterFS pracuje s lokálními i vzdálenými disky a redundanci poskytuje. Lze předpokládat, že tato funkcionalita vyžaduje určitou režii, díky které bude GlusterFS pomalejší.

Souborový systém ext4 byl vytvořen i připojen ve výchozím nastavení (`rw`, `relatime`, `data=ordered`). Všechny komponenty GlusterFS na všech uzlech byly ve verzi 3.5.2, pro měření byl použit svazek „flow“, který měl zapnuty optimalizační jednotky `write-behind`, `read-ahead`, `io-cache`, `quick-read`, `open-behind` a `md-cache`. Všechny testy byly spouštěny sekvenčně na jediném stroji. Čtení i zápis do ext4 znamená pouze lokální diskové operace, čtení z GlusterFS je prováděno lokálně, zápis je prováděn lokálně a zároveň na vzdálené disky následníka.

Test byly prováděny pomocí programu **Flexible I/O Tester**[1], který nabízí širokou škálu nastavení. Matice parametrů mého měření se ale skládala pouze z následující možností: směr (čtení, zápis), způsob přístupu (sekvenční, náhodný), velikost bloku (4 kB až 16 MB), použití/vynechání vyrovnávací paměti operačního systému (příznak `O_DIRECT`) a volba funkcí zprostředkovávajících vstupně/výstupní operace (synchronní `read()`/`write()` a asynchronní libaio). Každý test byl proveden minimálně třikrát, takže výsledný čas mohl být vypočítán jako průměr hodnot vzešlých z dílčích měření.

V grafu 8.2 jsou zobrazeny výsledky měření rychlosti sekvenčního a náhodného synchronního **čtení**. Vyrovnávací paměť operačního systému způsobila, že velikost bloku u sekvenčního čtení nehraje téměř žádnou roli, proto ext4 dosahuje pro všechny velikosti rychlosti přibližně 97 MB/s a GlusterFS mezi 65 a 75 MB/s. U náhodného přístupu je vyrovnávací paměť podstatně méně efektivní a malé velikosti bloků se negativně projevují na výsledném výkonu. Do 512 B je GlusterFS rychlejší, dále už ale vítězí ext4.

Výsledky obdobného měření rychlosti **zápisu** jsou vykresleny v grafu 8.3. Vyrovnávací paměť má pozitivní vliv pouze na ext4, při zápisu do GlusterFS se příliš neprojevuje a plného výkonu lze dosáhnou až od velikosti bloku 256 B. I pro větší bloky je ale zápis do ext4 zhruba dvakrát rychlejší.

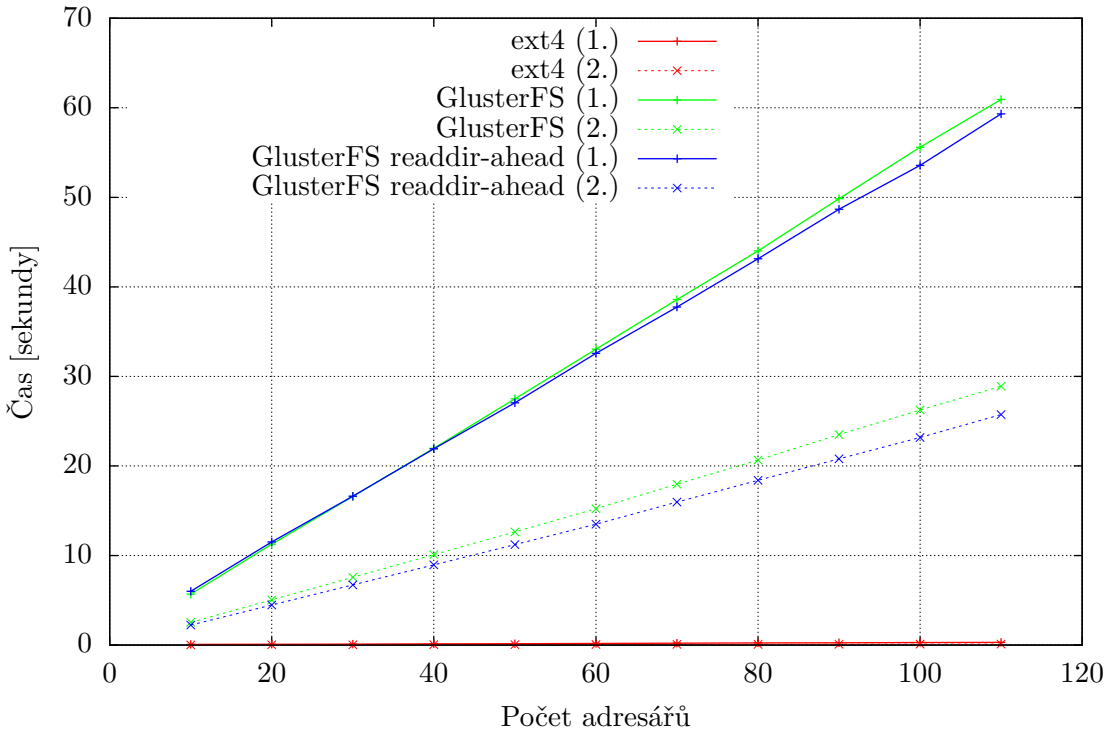
Pro srovnání přikládám také výsledky přímého synchronního čtení/zápisu (B.1 a B.2) a přímého asynchronního čtení/zápisu (B.3 a B.4). Rychlost zápisu do GlusterFS v obou případech klesla, čtení ale naopak při určitých velikostech bloku bylo dokonce rychlejší než ext4.

8.2.2 Průchod adresáři

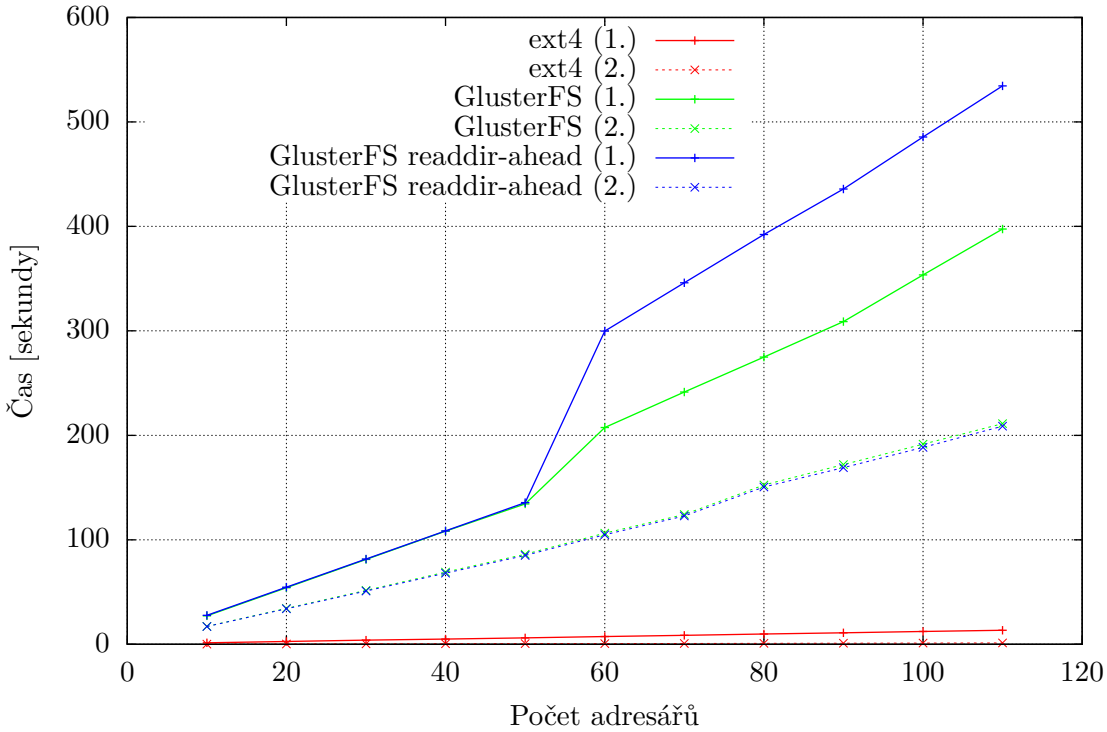
Tento experiment si klade za cíl srovnat souborové systémy ext4 a GlusterFS z pohledu rychlosti průchodu adresářem, v ostatním se od předchozího experimentu nijak neliší. I v tomto případě lze předpokládat, že GlusterFS bude pomalejší kvůli způsobu organizace metadat. Při adresářových operacích jako `opendir()`, `readdir()` nebo `closedir()` je totiž nutné kontaktovat všechny uzly obsažené v logickém svazku. Aby se tak nemuselo dít při každém volání funkce `readdir()`, nabízí GlusterFS možnost využít optimalizační jednotku `readdir-ahead`. Všechny testy jsem provedl s i bez ní, lze tak pozorovat její reálný vliv na výkon.

Základem testů bylo vytvoření **adresářové struktury**, pomocí které jsem se snažil simulovat způsob uložení záznamů o tocích. Různé hloubky adresářového stromu na výsledný čas neměly žádný vliv, proto jsem pro testování použil plochou strukturu o hloubce 1. Každý datový adresář tak byl přímý potomek testovacího kořenového adresáře, obsahoval 288 souborů (24 hodin rozdělených po pěti minutách) a každý soubor obsahoval několik

[1] https://github.com/axboe/fio



Obrázek 8.4: Měření odezvy průchodu adresáři příkazem `ls -R test_root`.



Obrázek 8.5: Měření odezvy průchodu adresáři a čtení prvních bajtů dat ze souborů příkazem `nfdump -I -R test_root`.

málo záznamů. Rychlost průchodu jsem měřil **příkazy** `ls -R test_root`, `find test_root` a `nfdump -I -R test_root`. Nástroje `ls` a `find` provádějí pouze rekurzivní průchod adresáři, `nfdump` navíc ještě čte několik prvních bajtů z každého souboru. Před prvním měřením byla na všech uzlech vyčištěna vyrovnávací paměť operačního systému pomocí příkazů `sync` a `echo 3 > /proc/sys/vm/drop_caches`. Následně bylo měření zopakováno, aby byl vidět vliv vyrovnávací paměti na výsledný čas.

Výsledky jsou dle očekávání, distribuovaný souborový systém je při práci s adresáři řádově pomalejší a `readdir-ahead` přináší pouze kosmetické zlepšení. Graf 8.4 zobrazuje výsledky měření doby odezvy příkazu `ls`, příkaz `find` podával téměř totožné výsledky a proto jej neuvádím. Ve všech případech doba odezvy rostla lineárně spolu s množstvím adresářů, na ext4 příkaz vždy podal výsledek v rámci několika milisekund, na GlusterFS to trvalo jednotky až desítky sekund. Druhý běh s naplněnou vyrovnávací pamětí zkrátil dobu zhruba na polovinu.

Odezvy příkazu `nfdump` jsou vykresleny v grafu 8.5. Časy se oproti `ls` v GlusterFS výrazně prodloužily, pravděpodobně díky nutnosti čtení prvních bajtů z každého souboru. Průběh se od předchozího měření liší skokem mezi 50 a 60 adresáři (na důvod jsem nepřišel) a také tím, že zapnutí optimalizace `readdir-ahead` mělo negativní vliv na výkon.

8.2.3 Rychlost synchronizace

GlusterFS má schopnost automatické synchronizace stavu po obnovení uzlu z výpadku (tzv. **automatické uzdravení**). Tento proces zahrnuje vyhledání rozdílů mezi originálem a replikou, přenesení rozdílových dat po síti a zapsání na disk opraveného uzlu. GlusterFS ale neumožňuje tento proces jakkoliv ovlivnit nebo nastavit jeho parametry, proto tento experiment zkoumá rychlost a spotřebu zdrojů procesu automatického uzdravení.

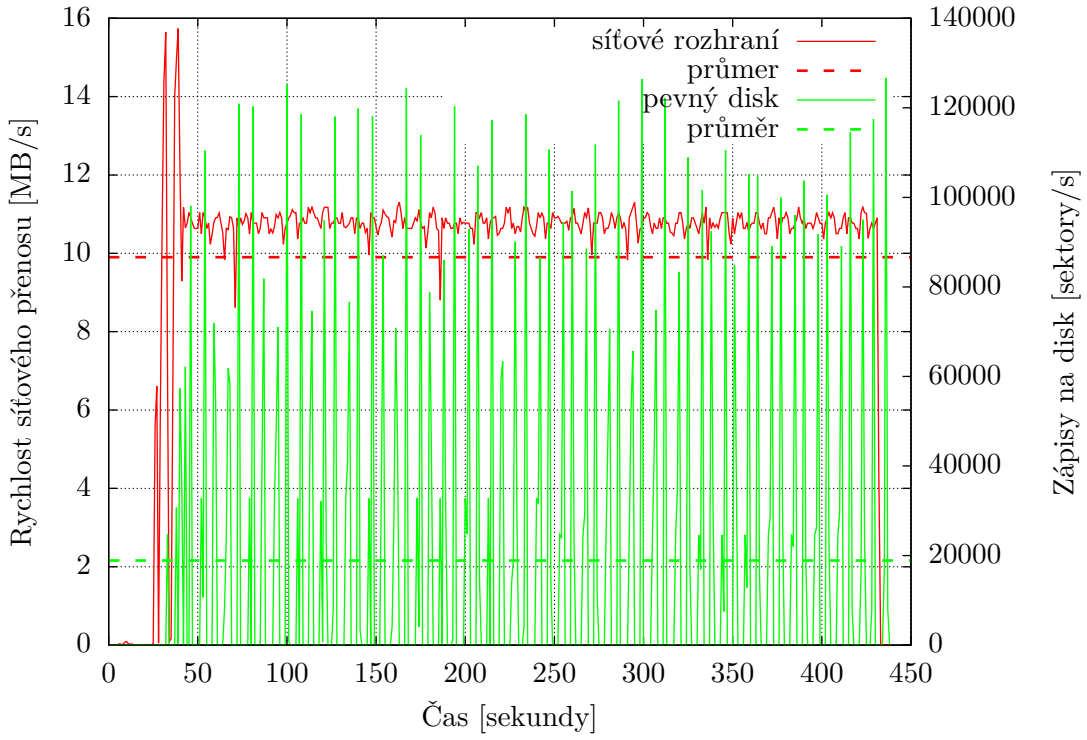
Výpadek jsem simuloval zapnutím brány firewall na uzlu A, který díky tomu nepřijímal ani neodesílal žádná data. Na uzlu B jsem v době výpadku vytvořil soubory obsahující pseudonáhodná data a po jejich úplném zapsání jsem vrátil do klastru uzel A. Od této doby až do stavu úplné synchronizace jsem na uzlu A sledoval množství dat přijatých na síťovém rozhraní a také počet zápisů na disk.

Při prvním měření (graf 8.6) jsem při výpadku vytvořil **jeden soubor o velikosti 4 GB**. Po vypnutí brány firewall na uzlu A trvalo nějakou dobu opětovné nastartování všech služeb (byly vypnuty díky fencingu), potom ale začala synchronizace téměř okamžitě. Soubor byl na uzel A přenesen za zhruba 440 sekund, což odpovídá průměrné rychlosti přenosu necelých 10 MB/s, ačkoliv teoretická propustnost spoje mezi uzly byla až 125 MB/s. Zápisy na disk jsou díky vyrovnávací paměti dávkového charakteru, jejich průměr je asi 20000 sektorů/s (jeden sektor má 512 B, zápisová rychlost je tedy přibližně 10,24 MB/s).

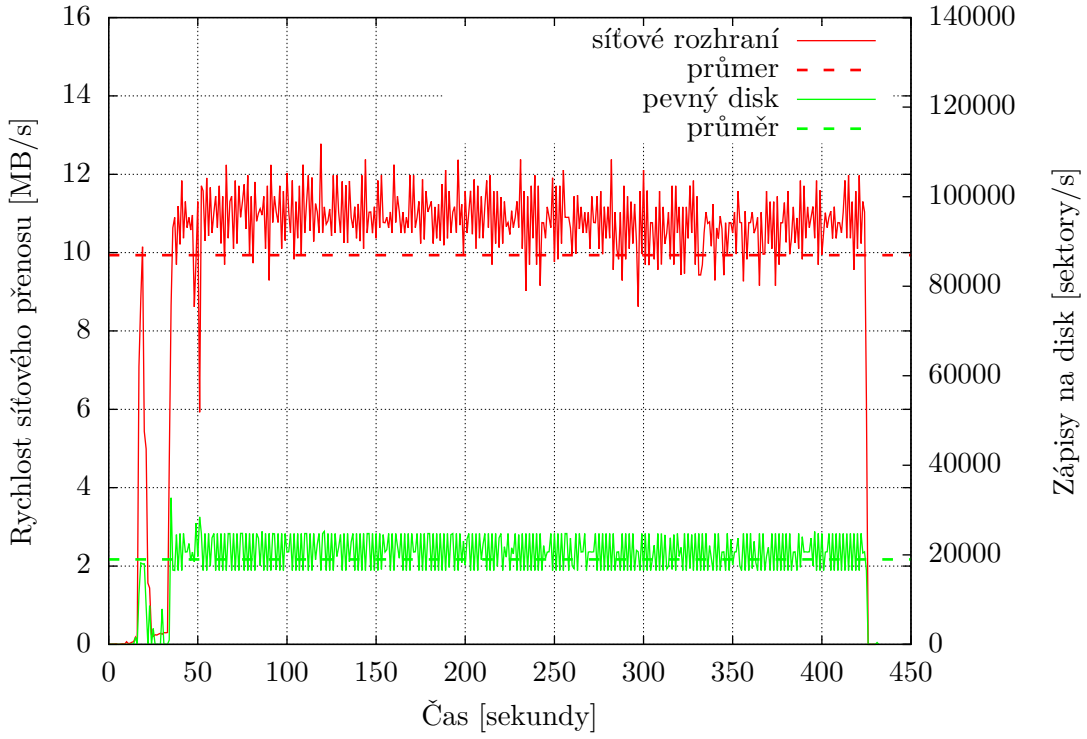
Druhé měření (graf 8.7) je obdobné, místo jednoho souboru jsem ale vytvořil **1000 souborů, každý o velikosti 4 MB**, aby byl celkový objem dat stejný jako v předchozím případě. Rozdíl je patrný především na způsobu zápisu na disk, který je mnohem častější. Průměrná rychlost přenosu dat zůstala přibližně stejná jako při prvním měření, menší soubory zjevně nedělají synchronizačnímu procesu problém.

8.3 Škálovatelnost dotazování

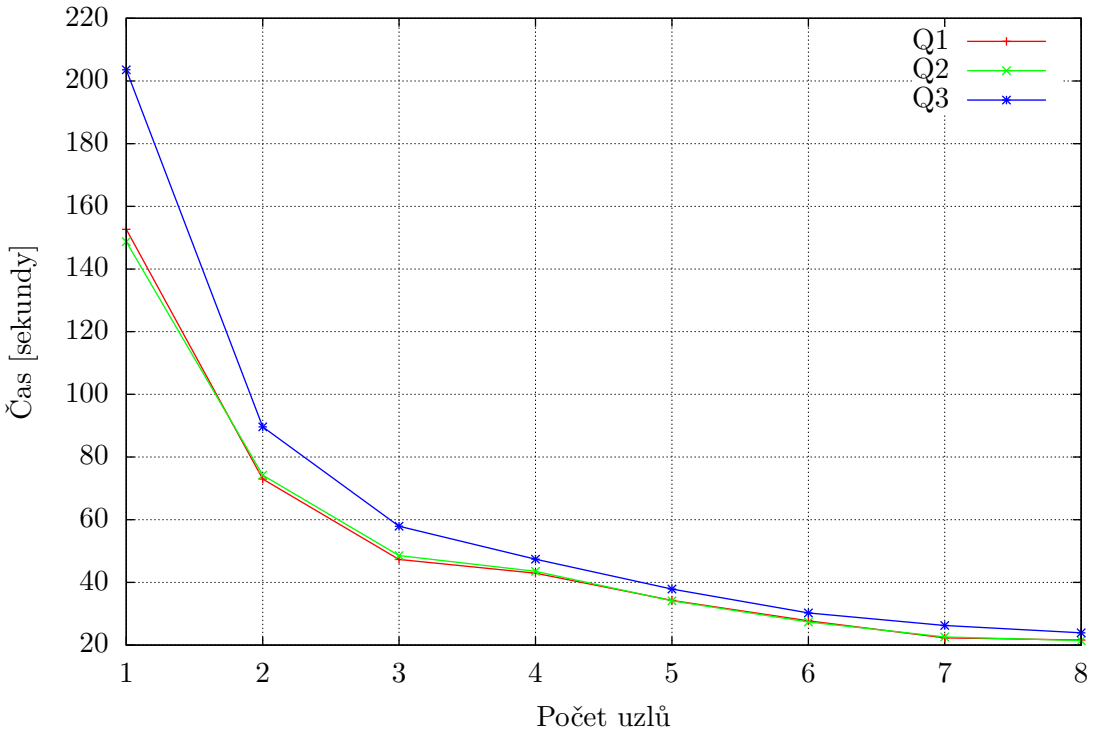
Poslední subsystém kolektoru, který je potřeba otestovat, je dotazování. To je v kolektoru zajištěno programem fdistdump, který jsem měl nainstalovaný ve verzi 0.1. Rozhodnul jsem se měřit škálovatelnost, tedy míru urychlení, které můžeme dosáhnout přidáním dalšího



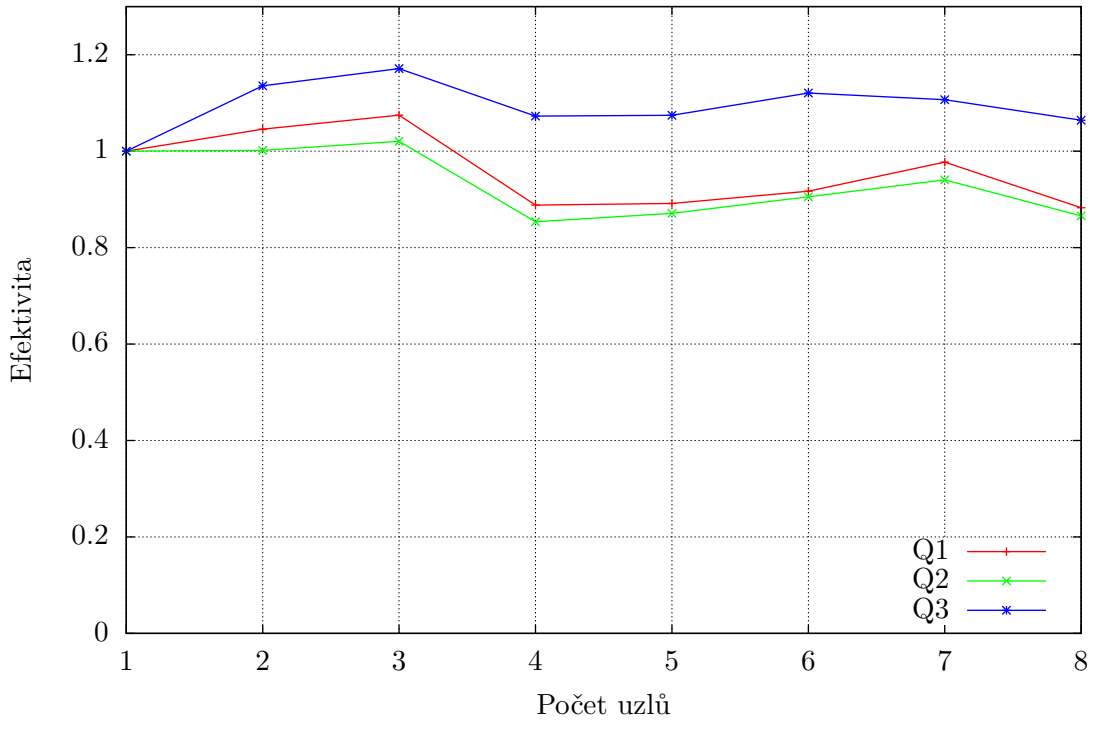
Obrázek 8.6: Rychlost synchronizace jednoho souboru o velikosti 4 GB.



Obrázek 8.7: Rychlost synchronizace tisíce souborů o velikosti 4 MB.



Obrázek 8.8: Doba běhu dotazů programu fdistdump v závislosti na počtu použitých uzlů.



Obrázek 8.9: Efektivita programu fdistdump v závislosti na počtu použitých uzlů.

uzlu. Klastr, který jsem používal pro předchozí experimenty má ale pouze čtyři uzly, což pro měření škálovatelnosti není ideální. Využil jsem proto jiný klastr o devíti uzlech, jeden z uzlů jsem používal jako řídící a při měření jsem jej do celkového počtu nezapočítával.

Měření jsem prováděl programem fdistdump na datové sadě obsahující 680 miliónu záznamů, které byly uloženy ve formátu nfdump a zabíraly zhruba 20 GB místa na disku. **Tři různé dotazy** Q1, Q2 a Q3 jsem postupně spouštěl až na osmi výpočetních uzlech a měřil dobu běhu programu. Před každým měřením jsem na všech uzlech vyčistil vyrovnávací paměť (stejným způsobem jako při experimentech s úložištěm), aby se soubory četly vždy z disku. Dotaz Q1 agreguje záznamy na základě čísla zdrojového portu, Q2 provádí oproti Q1 navíc ještě sestupné řazení podle počtu toků a vypisuje pouze prvních deset. Q3 provádí filtraci na port 53, agregaci podle IP adres, sestupné řazení podle počtu bajtů a vypisuje také prvních deset. Dotazy Q2 a Q3 jsou typu Top-N.

Graf 8.8 zobrazuje dobu běhu dotazů na jednom až na osmi uzlech. Zajímavější je ale **efektivita** (graf 8.9), kterou jsem vypočítal jako poměr zrychlení a počtu použitých uzlů. Standardně se efektivita pohybuje v rozsahu 0 až 1, v mém měření je ale místy větší než 1, objevuje tzv. superlineární zrychlení. To znamená, že běh na N uzlech je více než N-krát rychlejší než běh na jednom uzlu. Důvod tohoto chování přisuzuji malým vyrovnávacím pamětem na čipu procesoru: čím více uzlů, tím méně dat je potřeba zpracovat na jednom uzlu a tím větší procento dat se vměstná do procesorové cache.

8.4 Shrnutí výsledků

Experimenty s **poruchami a výpadky** přinesly pozitivní výsledky. Doba trvání detekce poruchy hardwaru je krátká a následné akce pro uvedení klastru do operativního stavu jsou vykonány v jednotkách sekund. Chování klastru při různých typech poruch bylo ověřeno jak pomocí softwarové simulace, tak zavedením skutečné hardwarové poruchy.

Výkonnostní testy **úložiště** v podobě DFS GlusterFS ukázaly, že cena za automatickou replikaci a synchronizaci po výpadku není malá. Při běžném synchronním čtení souborů se dostaneme maximálně na 75 % výkonu lokálního disku, při zápisu je to už pouze 55 %. Lépe nedopadlo ani procházení adresářů, které je oproti ext4 řádově pomalejší. Synchronizace po výpadku ale v rámci experimentů dopadla dobře, změny byly detekovány rychle a bez závislosti na velikosti/počtu souborů. Využití síťového rozhraní a lokálního úložiště při běhu synchronizace bylo přibližně 10 %, což mělo za následek delší trvání synchronizace, ale menší dopad na výkon klastru.

Změřena byla také škálovatelnost **dotazovacího subsystému** kolektoru. Na základních třech typech dotazu se ukázalo, že s každým dalším uzlem zahrnutým do úlohy roste celkový výkon lineárně a v některých případech je možné zaznamenat dokonce superlineární zrychlení.

Kapitola 9

Závěr

Cílem této práce bylo seznámit se s oblastí sběru dat o síťovém provozu na úrovni IP toků a nastudovat problematiku zpracování velkého množství dat o síťových tocích. Další oblastí studia byla vysoká dostupnost a spolehlivost v distribuovaném systému, konkrétně v distribuovaném kolektoru. Dostatečná znalost těchto témat měla vyústit v analýzu možností a následný návrh rozšíření distribuovaného kolektoru v oblasti spolehlivosti, dostupnosti a řízení dotazování. Úkolem také bylo navržená rozšíření implementovat a výsledek vyhodnotit pomocí vhodné sady měření a experimentů.

První část textu obsahuje informace o správě počítačových sítí a sledování síťového provozu. Jednotlivé metody jsou zmíněny od nejzákladnějších, v podobě SNMP a RMON, přes sběr dat na úrovni IP toků až po pokročilou hloubkovou analýzu paketů. Největší důraz v první části je kladen na monitorování síťových toků, které je pro tuto práci klíčové. Detailně jsou rozebrány fáze pro záchyt a předzpracování paketů na sondě, měření a export vzniklých záznamů, sběr a uložení dat na kolektoru a také poslední fáze v podobě analýzy dat.

Kombinace vysoké dostupnosti a distribuovaných systémů vyžaduje znalost obou těchto pojmů, proto jsou v následující části zmiňované pojmy rozebrány. U distribuovaných systémů jsou rozebrány jejich cíle, klady, zápory, nebo rozdíly oproti systémům centralizovaným, následně je obecně popsána vysoká dostupnost a jsou také objasněny pojmy jako spolehlivost nebo provozuschopnost. Tato část je zakončena praktičtějšími tématy, diskutovaný je CAP teorém aplikovaný na distribuovaný kolektor, problém volby kvóra, rozdělení sítě, nebo virtuální synchronie.

Na základě předchozí analýzy jsem mohl vytvořit návrh požadovaných rozšíření. Aby bylo možné tento text pochopit, bylo nutné popsat stav, ve kterém se distribuovaný kolektor nachází nyní. Následně se představují dvě architektury obohacující současný stav, první se vyznačuje vysokým výkonem a vysokou cenou, druhá cílí na nízký počet potřebných komponent a tedy i nižší cenu. V oblasti uložení dat se nabízí možnost použití sdílené množiny disků, nebo výhradně lokálních úložišť, obě možnosti jsou v práci přehledně rozděleny a podrobně popsány.

Ze dvou navrhovaných architektur jsem zvolil variantu, která cílí na nízký počet potřebných zařízení a při použití lokálních úložišť je navíc také bez sdílených komponent. Do současné architektury tak není potřeba přidávat žádný další hardware, přesto jsou eliminovány všechny body selhání. Žádný dodatečný hardware ale znamená nutnost řešit výpadky softwarově, proto jsem vytvořil programový zásobník složený z klastrové komunikační vrstvy, správce zdrojů a aplikačních programů. Komunikační vrstvu zajišťuje program Corosync, jako správce zdrojů je použit Pacemaker, o úložiště se stará souborový systém

GlusterFS, ukládací část realizuje IPFIXcol, a analytickou část obstarává program fdistdump. Jednotlivé programy bylo v mnoha případech nutné rozšířit nebo upravit, proto jsem implementoval potřebné změny, programy jsem nakonfiguroval a uvedl do provozu na zkušebním klastru.

Poslední část práce je věnovaná experimentům a výsledkům měření. Simulací poruch jsem si ověřil, že se klastr při skutečné poruše zachová správně a výpadek některého zdroje neovlivní celkovou dostupnost. Pozitivní výsledky přinesly také měření škálovatelnosti dotazovacího subsystému, jehož výkon spolu s každým dalším uzlem roste lineárně. Experimenty s úložištěm ověřily funkčnost automatické replikace a synchronizace dat po výpadku, také ale poukázaly na negativní dopad na výkon lokálních úložišť.

Práce přináší ucelenou a kompletní architekturu distribuovaného kolektoru bez sdílených komponent a bez jediného bodu selhání, přesto ale otevírá široké možnosti dalšího výzkumu a vývoje. Experimenty prokázaly negativní dopad GlusterFS na výkon, proto by bylo vhodné se této oblasti dále věnovat a pokusit se najít vhodnou alternativu. Také současný formát uložení dat kompatibilní s programem nfdump přestává být vyhovující a bude potřeba jej nahradit, inspiraci je možné hledat třeba ve sloupcových databázích.

Literatura

[1] Abadi, D. J.; Boncz, P. A.; Harizopoulos, S.: Column-oriented Database Systems. *Proc. VLDB Endow.*, ročník 2, č. 2, Srpen 2009: s. 1664–1665, ISSN 2150-8097.
URL http://dx.doi.org/10.14778/1687553.1687625

[2] Abadi, D. J.; Madden, S. R.; Hachem, N.: Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ACM, 2008, s. 967–980.

[3] ABZ knihy, a.s.: Kvorum – slovník cizích slov [online]. [cit. 10. 5. 2016].
URL http://slovnik-cizich-slov.abz.cz/web.php/slovo/kvorum-kvorum

[4] Amir, Y.; Moser, L. E.; Melliar-Smith, P. M.; aj.: The Totem Single-ring Ordering and Membership Protocol. *ACM Trans. Comput. Syst.*, ročník 13, č. 4, Listopad 1995: s. 311–342, ISSN 0734-2071.
URL http://doi.acm.org/10.1145/210223.210224

[5] Anderson, N.: Deep packet inspection meets 'Net neutrality, CALEA. *Ars Technica*, ročník 26, 2007.
URL http://arstechnica.com/gadgets/2007/07/deep-packet-inspection-meets-net-neutrality/

[6] Anthony, R. J.: *Systems Programming: Designing and Developing Distributed Applications*. Elsevier, první vydání, 2015, ISBN 978-0-12-800729-7.

[7] Armbrust, M.; Fox, A.; Griffith, R.; aj.: A View of Cloud Computing. *Commun. ACM*, ročník 53, č. 4, Duben 2010: s. 50–58, ISSN 0001-0782.
URL http://doi.acm.org/10.1145/1721654.1721672

[8] Bendrath, R.: Global technology trends and national regulation: Explaining Variation in the Governance of Deep Packet Inspection. In *International Studies Association Annual Convention*, Únor 2009, s. 15–18.

[9] Bendrath, R.; Mueller, M.: The end of the net as we know it? Deep packet inspection and internet governance. *New Media & Society*, ročník 13, č. 7, 2011: s. 1142–1160.
URL http://nms.sagepub.com/content/13/7/1142.abstract

[10] Birman, K.; Joseph, T.: Exploiting Virtual Synchrony in Distributed Systems. *SIGOPS Oper. Syst. Rev.*, ročník 21, č. 5, Listopad 1987: s. 123–138, ISSN 0163-5980.
URL http://doi.acm.org/10.1145/37499.37515

[11] Birman, K.; Schiper, A.; Stephenson, P.: Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, ročník 9, č. 3, Srpen 1991: s. 272–314, ISSN

0734-2071.
URL http://doi.acm.org/10.1145/128738.128742

[12] Bondi, A. B.: Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, New York, NY, USA: ACM, 2000, ISBN 1-58113-195-X, s. 195–203.
URL http://doi.acm.org/10.1145/350391.350432

[13] Brewer, E. A.: Towards robust distributed systems. In *PODC*, ročník 7, 2000.

[14] Case, J.; Fedor, M.; Schoffstall, M.; aj.: A Simple Network Management Protocol (SNMP). RFC 1157, RFC Editor, Květen 1990.
URL http://www.rfc-editor.org/rfc/rfc1157.txt

[15] Cisco Systems, I.: Cisco IOS NetFlow Version 9 Flow-Record Format. Technická zpráva, Cisco Systems, Inc., Květen 2011.
URL http://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9.pdf

[16] Cisco Systems, I.: Introduction to Cisco IOS NetFlow - A Technical Overview. Technická zpráva, Cisco Systems, Inc., Květen 2012.
URL http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html

[17] Ciuffoletti, A.: Monitoring a Virtual Network Infrastructure: An IaaS Perspective. *SIGCOMM Comput. Commun. Rev.*, ročník 40, č. 5, Říjen 2010: s. 47–52, ISSN 0146-4833.
URL http://doi.acm.org/10.1145/1880153.1880161

[18] Claise, B.; Trammell, B.: Information Model for IP Flow Information Export (IPFIX). RFC 7012, RFC Editor, Září 2013.
URL http://www.rfc-editor.org/rfc/rfc7012.txt

[19] Claise, B.; Trammell, B.; Aitken, P.: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, RFC Editor, Září 2013.
URL http://www.rfc-editor.org/rfc/rfc7011.txt

[20] Cluster Labs: Pacemaker [online]. [cit. 9. 5. 2016].
URL http://clusterlabs.org/

[21] Corp., S. G. I.: Technical Advances in the SGI UV Architecture. Technická zpráva, Silicon Graphics International Corp., Červen 2012.
URL https://www.sgi.com/pdfs/4192.pdf

[22] Coulouris, G.; Dollimore, J.; Kondberg, T.; aj.: *Distributed Systems: Concepts and Design*. Addison-Wesley, páté vydání, 2012, ISBN 978-0-13-214301-1.

[23] Dake, S. C.; Caulfield, C.; Beekhof, A.: The Corosync Cluster Engine. In *Linux Symposium*, ročník 85, Citeseer, Červenec 2008.

[24] Deri, L.; Lorenzetti, V.; Mortimer, S.: *Traffic Monitoring and Analysis: Second International Workshop, TMA 2010, Zurich, Switzerland, April 7, 2010.Proceedings*, kapitola Collection and Exploration of Large Data Monitoring Sets Using Bitmap Databases. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-12365-8, s. 73–86.
URL http://dx.doi.org/10.1007/978-3-642-12365-8_6

[25] Donvito, G.; Marzulli, G.; Diacono, D.: Testing of several distributed file-systems (HDFS, Ceph and GlusterFS) for supporting the HEP experiments analysis. In *Journal of Physics: Conference Series*, ročník 513, IOP Publishing, 2014, str. 042014.

[26] European Parliament and of the Council: Directive 2006/24/EC. Březen 2006.
URL
http://eur-lex.europa.eu/legal-content/EN/NOT/?uri=CELEX:32006L0024

[27] Flowmon Networks, a.s.: FlowMon 7.0 new features and benefits [online]. [cit. 28.12.2015].
URL https://www.invea.com/data/flowmon/flowmon-7_0-news.pdf

[28] Forum, M. P. I.: MPI: A Message-Passing Interface Standard, Version 3.1. Technická zpráva, Červen 2015.

[29] Frank, J.: Artificial intelligence and intrusion detection: Current and future directions. In *Proceedings of the 17th national computer security conference*, ročník 10, Baltimore, MD, 1994, s. 1–12.

[30] Froom, R.; Sivasubramanian, B.; Frahim, E.: *Implementing Cisco IP Switched Networks (SWITCH)*. Cisco Press, první vydání, 2010, ISBN 978-1-58705-884-4.

[31] GlusterFS: GlusterFS Documentation [online]. [cit. 9. 5. 2016].
URL http://gluster.readthedocs.io/en/latest/

[32] Grosu, D.; Chronopoulos, A. T.; Leung, M.-Y.: Load balancing in distributed systems: an approach using cooperative games. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, April 2002, s. 10 pp–, doi:10.1109/IPDPS.2002.1015536.

[33] Haas, F.: Ahead of the Pack: The Pacemaker High-availability Stack. *Linux J.*, ročník 2012, č. 216, Duben 2012, ISSN 1075-3583.
URL http://dl.acm.org/citation.cfm?id=2208859.2208863

[34] Hofstede, R.; Sperotto, A.; Fioreze, T.; aj.: *Networked Services and Applications - Engineering, Control and Management: 16th EUNICE/IFIP WG 6.6 Workshop, EUNICE 2010, Trondheim, Norway, June 28-30, 2010. Proceedings*, kapitola The Network Data Handling War: MySQL vs. NfDump. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-13971-0, s. 167–176.
URL http://dx.doi.org/10.1007/978-3-642-13971-0_16

[35] Hofstede, R.; Čeleda, P.; Trammell, B.; aj.: Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *Communications Surveys Tutorials, IEEE*, ročník 16, č. 4, 2014: s. 2037–2064, ISSN 1553-877X.

[36] IBM: Improving Systems Availability. Technická zpráva, Září 1998.
URL http://www.dis.uniroma1.it/irl/docs/availabilitytutorial.pdf

[37] ISO/IEC: 10746-3:2009 Information technology – Open distributed processing – Reference model: Architecture. ISO, International Organization for Standardization/International Electrotechnical Commission, Geneva, Switzerland, 2009.

[38] Košnar, T.: Notes to Flow-Based Traffic Analysis System Design. Technická zpráva, CESNET z.s.p.o., Prosinec 2004.
URL http://archiv.cesnet.cz/doc/techzpravy/2004/ftas-design/

[39] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, ročník 21, č. 7, Červenec 1978: s. 558–565, ISSN 0001-0782.
URL http://doi.acm.org/10.1145/359545.359563

[40] Lee, I.: Introduction to Distributed Systems [online]. Slidy k přednášce, 2007, [cit. 5. 5. 2016].
URL http://www.cis.upenn.edu/~lee/07cis505/Lec/lec-ch1-DistSys-v4.pdf

[41] Lee, Y.; Lee, Y.: Toward Scalable Internet Traffic Measurement and Analysis with Hadoop. *SIGCOMM Comput. Commun. Rev.*, ročník 43, č. 1, Leden 2012: s. 5–13, ISSN 0146-4833.
URL http://doi.acm.org/10.1145/2427036.2427038

[42] Li, K.; Hudak, P.: Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, ročník 7, č. 4, Listopad 1989: s. 321–359, ISSN 0734-2071.
URL http://doi.acm.org/10.1145/75104.75105

[43] Linux-HA: Open Source High-Availability Software for Linux [online]. [cit. 9. 5. 2016].
URL http://www.linux-ha.org/wiki/Main_Page

[44] Modiri, R.; Moiin, H.: System and method for determining cluster membership in a heterogeneous distributed system. Únor 2001, US patent 6192401.
URL https://www.google.com/patents/US6192401

[45] Morken, J. T.: Distributed netflow processing using the map-reduce model. 2010.
URL http://hdl.handle.net/11250/252140

[46] Moser, L. E.; Amir, Y.; Melliar-Smith, P. M.; aj.: Extended virtual synchrony. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, Červen 1994, s. 56–65.

[47] Porter, T.: The perils of deep packet inspection. *Security Focus*, 2005.
URL
http://www.symantec.com/connect/articles/perils-deep-packet-inspection

[48] Postel, J.: User Datagram Protocol. RFC 2819, RFC Editor, Srpen 1980.
URL http://www.rfc-editor.org/rfc/rfc768.txt

[49] Puš, V.; Kekely, L.; Špinler, M.; aj.: HANIC 100G: Hardware accelerator for 100 Gbps network traffic monitoring. Technická zpráva, CESNET, Únor 2014.
URL https://www.cesnet.cz/wp-content/uploads/2015/01/hanic-100g.pdf

[50] Resman, M.: *CentOS High Availability*. Packt Publishing, první vydání, 2015, ISBN 978-1785282485.

[51] Schmidt, K.: *High Availability and Disaster Recovery: Concepts, Design, Implementation*. Springer, první vydání, 2006, ISBN 978-3-540-24460-8.

[52] Silberschatz, A.; Korth, H. F.; Sudarshan, S.: *Database system concepts*. McGraw-Hill New York, šesté vydání, 2010, ISBN 0-07-352332-1.

[53] Stallings, W.: *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., třetí vydání, 1998, ISBN 0201485346.

[54] Steinberger, J.; Schehlmann, L.; Abt, S.; aj.: Anomaly Detection and Mitigation at Internet Scale: A Survey. In *Emerging Management Mechanisms for the Future Internet*, *Lecture Notes in Computer Science*, ročník 7943, Springer Berlin Heidelberg, 2013, ISBN 978-3-642-38997-9, s. 49–60.
URL http://dx.doi.org/10.1007/978-3-642-38998-6_7

[55] Stewart, R.: Stream Control Transmission Protocol. RFC 4960, RFC Editor, Září 2007.
URL http://www.rfc-editor.org/rfc/rfc4960.txt

[56] Stewart, R. and Ramalho, M, and Xie, Q. and Tuexen, M. and Conrad, P.: Stream Control Transmission Protocol (SCTP): Partial Reliability Extension. RFC 3758, RFC Editor, Květen 2004.
URL http://www.rfc-editor.org/rfc/rfc3758.txt

[57] Stonebraker, M.: The case for shared nothing. *IEEE Database Eng. Bull.*, ročník 9, č. 1, 1986: s. 4–9.

[58] Tanenbaum, A. S.: *Distributed systems: Principles and Paradigms*. Pearson, druhé vydání, 2007, ISBN 0-13-239227-5.

[59] Taylor, Z.; Ranganathan, S.: *Designing High Availability Systems: DFSS and Classical Reliability Techniques with Practical Real Life Examples*. Wiley-IEEE Press, první vydání, 2014, ISBN 978-1118551127.

[60] University of Southern California: Transmission Control Protocol. RFC 2819, RFC Editor, Září 1981.
URL http://www.rfc-editor.org/rfc/rfc793.txt

[61] Velan, P.: Practical experience with IPFIX flow collectors. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, Květen 2013, ISSN 1573-0077, s. 1021–1026.

[62] Velan, P.; Krejčí, R.: Flow information storage assessment using IPFIXcol. In *Dependable Networks and Services*, Springer, 2012, s. 155–158.

[63] Waldbusser, S.: Remote Network Monitoring Management Information Base. RFC 2819, RFC Editor, Květen 2000.
URL http://www.rfc-editor.org/rfc/rfc2819.txt

[64] Wikipedia: init — Wikipedia, The Free Encyclopedia [online]. 2016, [cit. 18.5.2016].
URL https://en.wikipedia.org/wiki/Init

[65] Wrona, J.: Identifikace aplikačních protokolů. 2014, bakalářská práce.

[66] Zhang, J.; Moore, A.: Traffic Trace Artifacts due to Monitoring Via Port Mirroring. In *End-to-End Monitoring Techniques and Services, 2007. E2EMON '07. Workshop on*, 2007, s. 1–8.

[67] Zittrain, J. L.; Edelman, B. G.: Internet filtering in China. *IEEE Internet Computing*, Březen 2003.

[68] Zseby, T.; Molina, M.; Duffield, N.; aj.: Sampling and Filtering Techniques for IP Packet Selection. RFC 5475, RFC Editor, Březen 2009.
URL http://www.rfc-editor.org/rfc/rfc5475.txt

[69] Žádník, M.; Kořenek, J.; Puš, V.; aj.: Distribuovaný kolektor záznamů o IP tocích: návrh a první experiment. Technická zpráva, CESNET z.s.p.o., Prosinec 2014.
URL https://www.cesnet.cz/wp-content/uploads/2015/04/securitycloud.pdf

Přílohy

Seznam příloh

Příloha A

Obsah CD

V kořenovém adresáři se nachazí adresáře s následujícím obsahem:

corosync: konfigurační soubor corosync.conf

pacemaker: konfigurační XML pro CIB, konfigurační soubor pro crmsh

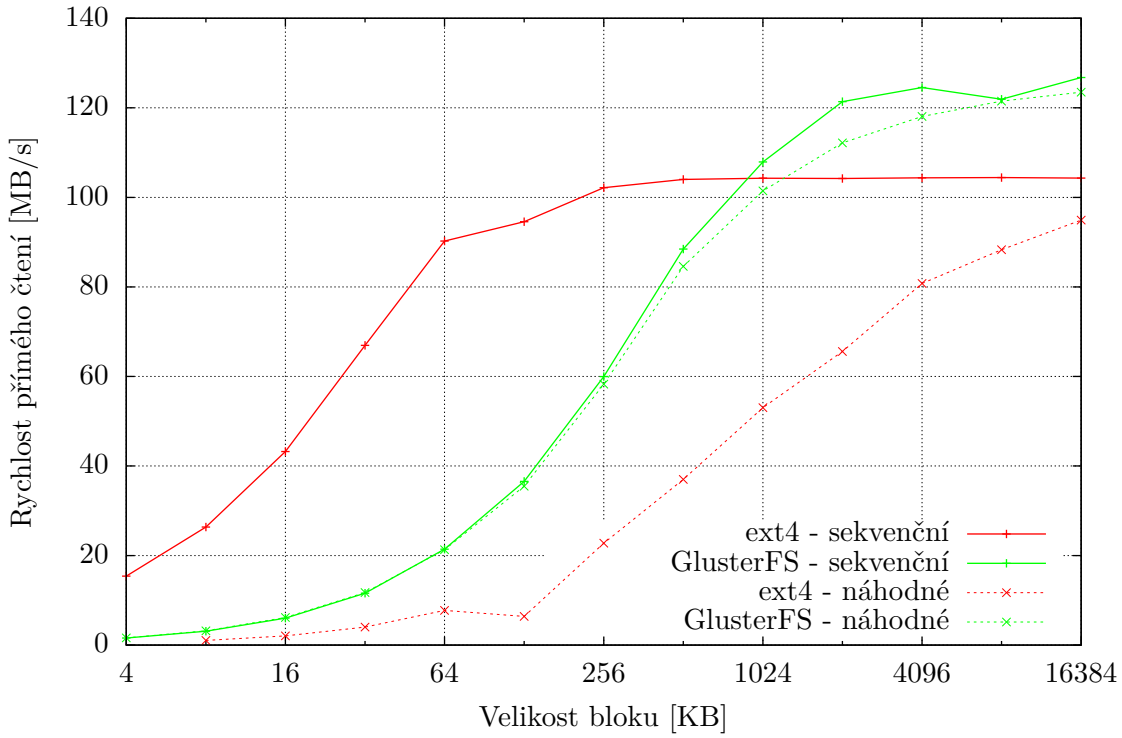
glusterfs: filter skript pro správné fungování NUFA při více než jedné cihle na jednom uzlu, konfigurační volume soubory pro svazky „conf“ a „flow“

ipfixcol: zdrojový soubor UDP-CPG vstupního pluginu, skript a metadata aplikačního agenta

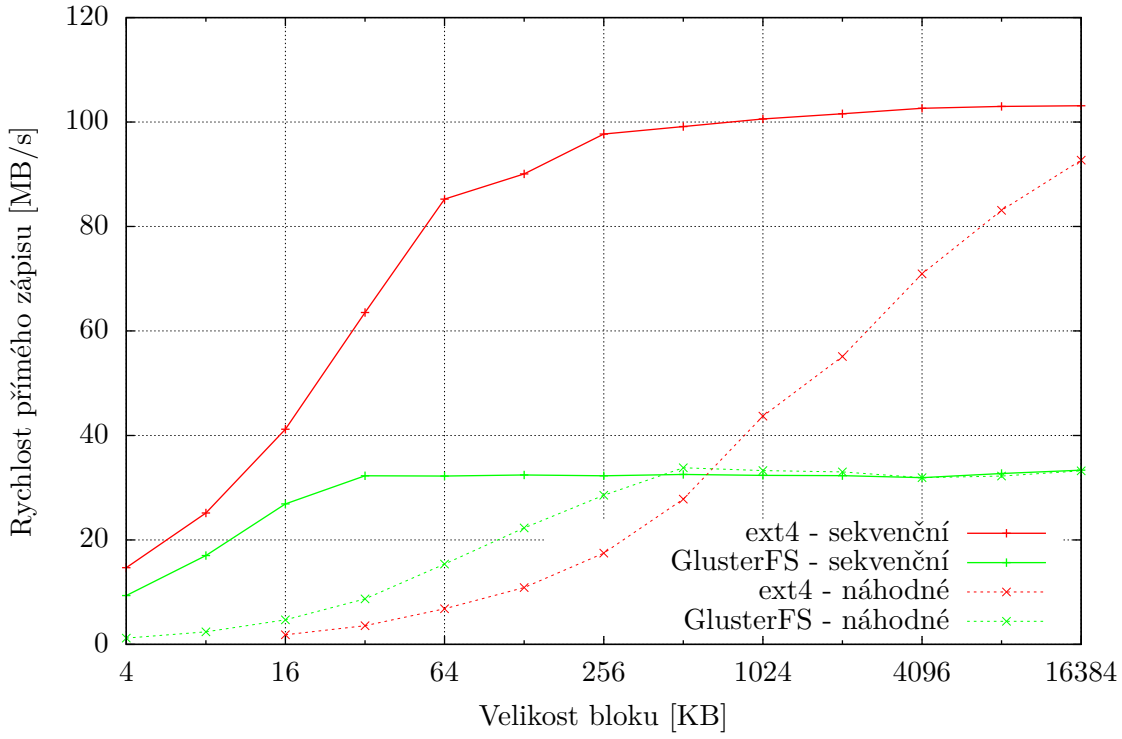
fdistdump: zdrojové soubory programu včetně obalovacího spouštěcího skriptu

Příloha B

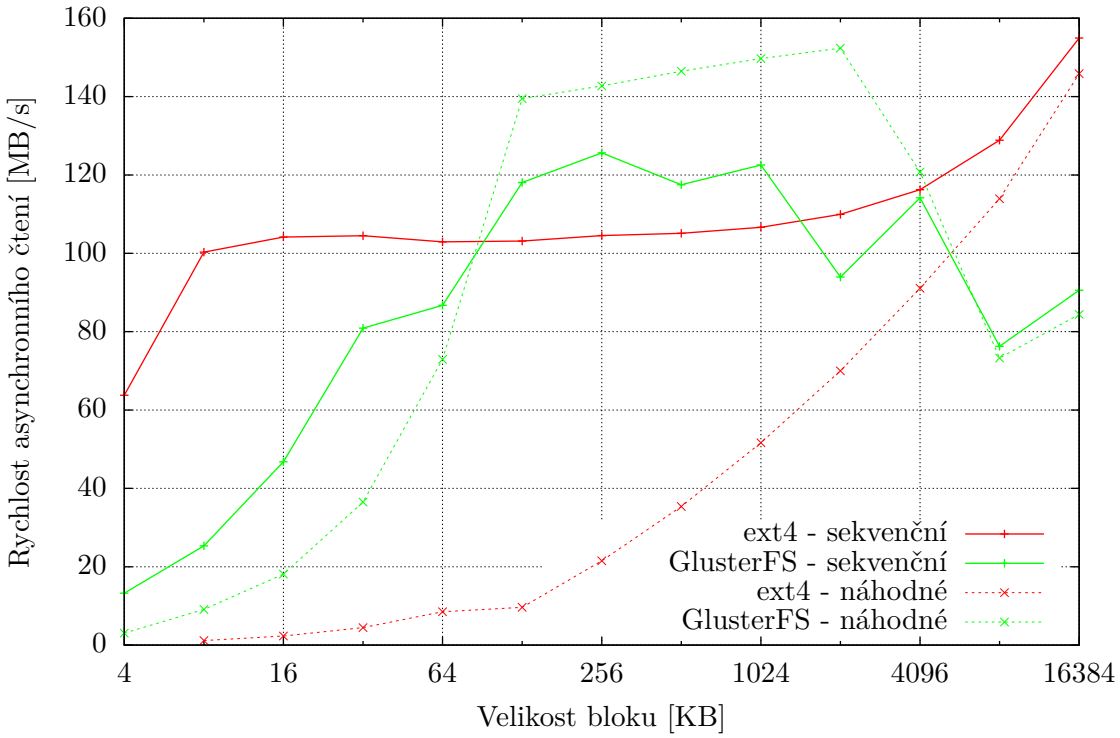
Dodatkové výsledky měření rychlosti čtení a zápisu



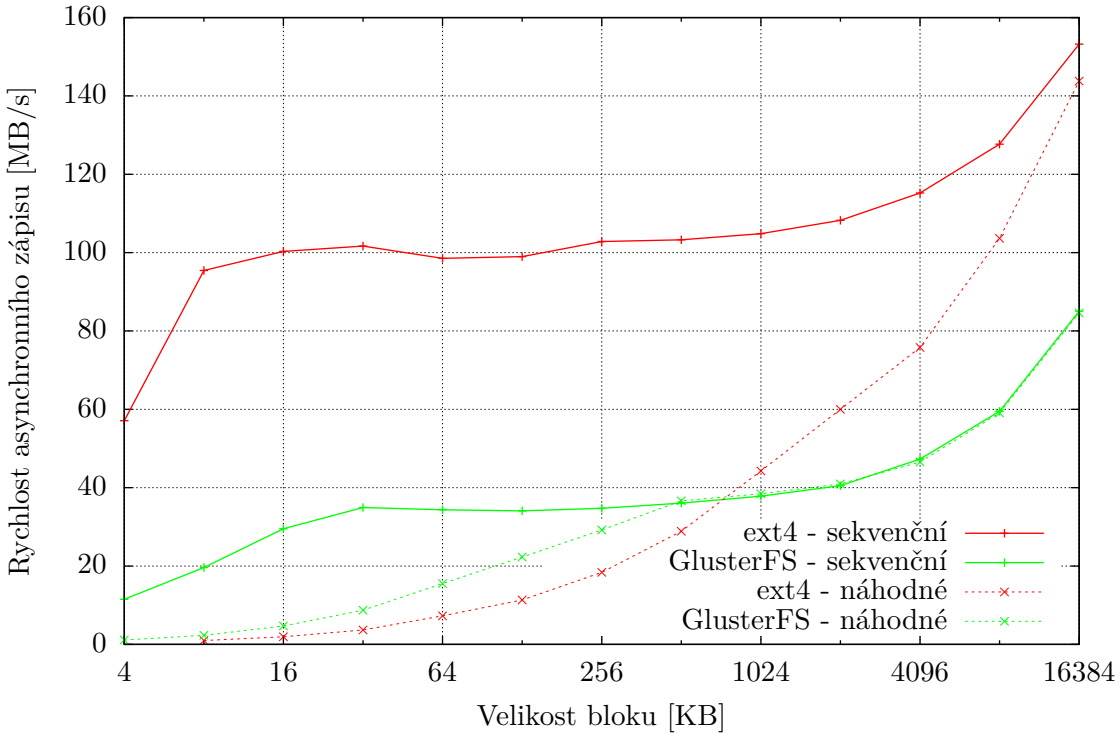
Obrázek B.1: Rychlost synchronního čtení bez použití vyrovnávací paměti.



Obrázek B.2: Rychlost synchronního zápisu bez použití vyrovnávací paměti.



Obrázek B.3: Rychlost asynchronního čtení bez použití vyrovnávací paměti.



Obrázek B.4: Rychlost asynchronního zápisu bez použití vyrovnávací paměti.